

2011

Algoritmos y estructuras
de datos

Pablo Nogueira

[GUIÓN DE CLASES TEÓRICAS]

El guión de clases teóricas de algoritmos y estructuras de datos del profesor Pablo Nogueira que complementan las clases, el libro básico y las transparencias. Adaptado a esta versión por el alumno Pau Arlandis.

Algoritmos y Estructura de Datos

Guión de clases teóricas del Profesor Pablo Nogueira

Adaptado a doc, pdf y ePub por Pau Arlandis

Este fichero es un guión que se complementa en clase con transparencias, explicaciones en la pizarra y discusiones.

Las secciones y subsecciones en general siguen el orden de secciones del libro. Los corchetes encierran referencias al libro, al código o a las transparencias. Los dobles corchetes encierran referencias al texto dentro del fichero que todavía no existen (existirán en el futuro). Se pone texto en **negrita** para enfatizarlo. Existen vínculos a zonas del propio texto o a páginas externas que permiten enlazar o aumentar conocimientos.

Los errores y erratas en el libro y las transparencias se indican en párrafos que comienzan con la palabra 'Errores'. También se proponen cuestiones y ejercicios (señalados con la palabra '**Ejercicio**') cuya resolución ayudará al alumno a comprender mejor y, en el caso de ejercicios avanzados, a profundizar en los contenidos de la asignatura. Algunos o parte de estos ejercicios se utilizarán como preguntas de examen.

Contenido

Guión de clases teóricas del Profesor Pablo Nogueira	2
Adaptado a doc, pdf y ePub por Pau Arlandis.....	2
Viernes 11-02-2011	6
Material de la asignatura	6
Fechas importantes:.....	6
Conceptos de Java y POO	6
Conceptos de Java y POO que los alumnos deben repasar	6
Viernes 18-02-2011	8
Abstracción, estructura de datos, y algoritmos	8
6.1 Array Lists	8
Material	8
Interfaz <code>IndexList<E></code>	8
Clase <code>ArrayIndexList<E></code>	9
Complejidad y costes amortizados.....	10
Interfaz <code>java.util.ArrayList</code> de la JCF	10
6.2 Node Lists	10

Material	10
Repaso del concepto de lista simple y doblemente enlazada	11
Interfaz Position<E>	11
Clase DNode<E>	11
Interfaz PositionList<E>	12
Clase NodePositionList<E>	12
Viernes 25-02-2011	14
6.3 Iterators	14
Material	14
Concepto de Iterador	14
Interfaces Iterator<T> e Iterable<T>	14
6.3.4 List Iterators in Java	21
6.4.2 Sequences	22
4.2 Analysis of Algorithms	22
Material	22
Resumen de principios fundacionales	22
Notación O()	23
Notaciones Omega y Theta	24
Complejidad: comentarios y ejercicios	24
6.4 List ADTs and the Collections Framework	24
Viernes 04-03-2011	25
7 Tree Structures	25
Material	25
7.1 General Trees	25
7.2 Tree Traversal Algorithms	30
7.2.2 Preorder Traversal	31
7.2.3 Postorder Traversal	31
Traversals: comentarios y ejercicio	32
7.3 Binary Trees	32
Viernes 11-03-2011	37
7 Tree Structures (continuación)	37
7.3 Binary Trees (continuación)	37
8 Priority Queues	39
Material	39

8.1 The Priority Queue Abstract Data Type	39
8.2 Implementing a Priority Queue with a List	44
Viernes 01-04-2011	46
8.3 Heaps.....	46
Material	46
Motivación de los montículos	46
Arboles Binarios (Casi)Completo	47
Montículos.....	52
8.3.3 Implementing a Priority Queue with a Heap	52
8.3.4 A Java Heap Implementation	53
8.3.5 Heap Sort.....	54
Viernes 08-04-2011	54
9.1 Maps.....	54
Material	54
Motivación de las funciones finitas.....	54
Definición matemática de función finita.....	55
9.1.1 The Map ADT	56
9.1.2 A Simple List-Based Map Implementation.	57
Maps en la JCF	59
9.2 Hash tables	60
Material	60
Motivación de tablas de dispersión	60
Definición de tablas de dispersión	60
9.2.3 Hash Codes.....	61
9.2.4 Compression Functions	62
9.2.5 Collision-Handling Schemes	63
9.2.7 Load Factors and Rehashing.....	67
9.2.6 A Java Hash Table Implementation	67
Viernes 15-04-2011	68
9.5 Dictionaries.....	68
Material	68
Motivación de los diccionarios.....	68
9.5.1 The Dictionary ADT.....	69
9.5.3 An Implementation Using the java.util Package	70

Viernes 06-05-2011	72
9.3 Ordered Maps	72
Material	72
Motivación de las funciones finitas con dominio ordenado	72
9.3.1 Ordered Search Tables and Binary Search	73
10.1 Binary Search Trees	75
Material	75
Motivación de los árboles binarios de búsqueda.	76
Representación mediante árboles binarios	76
10.1.1 Searching	77
10.1.2 Update Operations	78
10.1.3 Java Implementation	81
Críticas a la implementación BinarySearchTreeMap	84
Viernes 13-05-2011	85
10.2 AVL Trees	85
Material	85
Motivación de los árboles AVL	85
Representación mediante árboles binarios de búsqueda	85
10.2.1 Update Operations	86
10.2.2 Java Implementation	90
Críticas a la implementación AVLTreeMap.	92
Viernes 20-05-2011	92
10.4 (2,4) Trees	92
Material	92
Motivación de árboles multcamino de búsqueda y árboles (2,4)	92
10.4.1 Multi-Way Search Trees	93
Definición de árboles (2,4)	95
10.4.2 Update Operations for (2,4) Trees	95
Viernes 27-05-2011	98
14.2 External Memory and Caching	98
Material	98
Algunas notas a las transparencias	98
14.3 External Searching and B-Trees	99
Material	99

Motivación de los árboles (a,b) y B	99
14.3.1 (a,b) Trees.....	99
14.3.1 B-Trees	100

Viernes 11-02-2011

Material de la asignatura

- [Libro](#), 5ª edición.

Y además:

- [Código](#)
- [Javadoc](#)
- [Código con adiciones y transparencias](#)

Bibliografía complementaria recomendada y ocasionalmente citada en este guión:

- [JLS'3E] [Java Language Specification](#), 3rd Edition.
- [JPSE6] [Java Platform SE 6](#), incluye Java Collections Framework (JCF).
- [CLRS'01] Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein, "[Introduction to Algorithms](#)", 2nd Edition, MIT Press. (Hay una tercera edición.)
- [JP'05] Peter Sestoft, "[Java Precisely](#)", 2ed, MIT Press.
- [ALT'03] Moshe Augenstein, Yedidyah Langsam, Aaron M. Tenenbaum, "[Data Structures Using Java](#)", Prentice Hall.
- [AHU'83] Alfred Aho, John Hopcroft, Jeffrey Ullman, "[Data Structures and Algorithms](#)", Addison-Wesley. (Un clásico de los 80.)

Fechas importantes:

Primer Test	Viernes 11 de marzo de 2011	Semana 5
Segundo Test	Jueves 28 de abril de 2011	Semana 10
Entrega práctica	Semana del 16 al 20 de mayo	Semana 13
Tercer Test	Martes 14 de junio de 2011	Semana 17
Final Junio	Martes 14 de junio de 2011	Semana 17
Extraordinario	Lunes 4 de julio de 2011	Semana 20

Conceptos de Java y POO

Conceptos de Java y POO que los alumnos deben repasar

- Paquetes y compilación

- **Clases, atributos (fields) y métodos**
 - Clases, clases abstractas e interfaces.
 - Constructores.
 - Modificadores de visibilidad y de uso
 - Modificadores de clase, atributo, métodos y parámetros (public, private, protected, final, abstract, static, etc).
- **Tipos**
 - Básicos: void, boolean, char, byte, short, int, long, float, double.
 - Variables, referencias y objetos.
 - Conversión automática, casting y autoboxing.
 - Declaraciones e inicializaciones. Valores por defecto.
 - Arrays. Inicialización y creación.
- **Expresiones**
 - Aritméticas, lógicas, asignación, incrementos.
 - Asociación y precedencia de operadores.
- **Comandos**
 - Condicionales (if, if-else, switch).
 - Bucles (while, for, do-while, for-each).
 - Expresiones seguidas de punto y coma.
- **Herencia**
 - Subclase y superclase. Subclase y subtipo.
 - Herencia simple y múltiple. Interfaces (Mixins).
 - Variables this y super.
 - Variable polimórfica y enlazado dinámico.
 - Extensión, especialización, sobrescritura (reemplazo, refinamiento).
- **Sobrecarga**
 - Constructores, métodos y atributos.
 - Resolución de conflicto: por ámbito, por signatura.
- **Genéricos**
 - -Clases, métodos, atributos y arrays genéricos.
 - Interacción entre genéricos, sobrecarga y herencia.
 - Compilación en Java mediante borrado de tipos (type erasure).
- **Excepciones**
 - Checked y Unchecked.
 - Declaración (throws).
 - Lanzamiento (throw).
 - Captura (try-catch-finally).

Viernes 18-02-2011

Abstracción, estructura de datos, y algoritmos

- [Libro: 2.1.1 *Object-Oriented Design Goals*, p58-62]

Tipo abstracto de datos (TAD): interfaz (qué) e implementación (cómo). [Libro, p166]: "A systematic way of organizing and accessing data"

Abstracción, encapsulación, modularidad, y organización jerárquica.

Algoritmos: usados en implementación y uso de TADs. [Libro, p166]: "a step-by-step procedure for performing some task in a finite amount of time".

6.1 Array Lists

Material

- Libro: sección 6.1
- Transparencias: arraylists.pdf
- Código:
 - IndexList.java (interfaz), ArrayIndexList.java (implementación).
 - Generic_array_creation (Código explicativo sobre vectores genéricos).
- ERRORES:
 - Transparencias y Libro: El texto llama 'Array List' al TAD pero el nombre del interfaz Java es 'IndexList'. Tiene más sentido que así se llame el TAD ya que el uso de arrays es una cuestión de implementación.
 - Transparencia 2:
 - El TAD no 'extiende' la noción de array: la abstrae.
 - Los índices no tienen tipo 'Integer' sino 'int'.
 - **Ejercicio:** ¿Sería mejor tener 'Integer'?
 - La clase de los elementos no es 'Object' sino 'E'.

Interfaz IndexList<E>

Motivación: *abstracción de datos.*

- Tenemos arrays en Java, ¿Para qué un TAD?
- Respuesta: métodos 'add' y 'remove' + estirar y encoger el array.

Características a resaltar del TAD:

- Tamaño del array: N.
- Número de elementos almacenados: n.
- Los elementos se almacenan en posiciones 0 a n-1 sin huecos vacíos entre elementos.
- Rango ('rank') vs Rango ('range'):
 - El rango ('range') es el intervalo [0..n-1].
 - El rango ('rank'): un elemento en el índice i está en el rank i+1.

- El interfaz `IndexList<E>` utiliza la variable 'i' para índices.
 - o La implementación `ArrayIndexList<E>` utiliza la variable 'r' para índices.
 - o NO se usan ranks.
- Se accede a los elementos usando índices:
 - o Índices para 'get', 'set', y 'remove' entre 0 y n-1 inclusive.
 - o Índice para 'add' entre 0 y n inclusive. ¿Por qué? → Se puede añadir al final.
 - o Se lanza excepción '`IndexOutOfBoundsException`' si no se cumple.
- **Ejercicio:** *¿Por qué 'set' devuelve el elemento que es reemplazado?*
 - o (Pista: composición de invocaciones de métodos.)
- Déficit ('underflow'): Cuando n cae por debajo de una cota, por ejemplo, $n < N/4$. Se puede encoger el TAD (que no el array). No llamamos déficit a cuando se invoca 'get' con el TAD vacío, esto se controla con '`IndexOutOfBoundsException`'.
- Desbordamiento ('overflow'): Se invoca 'add' dejando el array lleno. Al invocar 'add' inmediatamente después se produce un desbordamiento lanzándose la excepción '`IndexOutOfBoundsException`'.
Se puede alargar el TAD (que no el array). Incremento constante o proporcional al tamaño original. [Transparencias 9-11, sobre costes amortizados. Libro, p241-242]. Ver [Complejidad y costes amortizados](#)
- `IndexList.java` importa el interfaz '`java.util.Iterator`' pero no se define un método '`iterator`' que devuelva un iterador y tampoco se extiende del interfaz `Iterable<E>`. Ver [6.3 Iterators](#).
- **Ejercicio:** *¿Qué habría que hacer para poder iterar sobre los elementos de un `IndexList<E>`?*

[Libro, 6.1.2 *The Adapter Pattern*, p235]: esta sección explica que `IndexList<E>` se puede adaptar (patrón de diseño Adapter) para definir el interfaz `Deque<E>` que no hemos visto.

Clase `ArrayIndexList<E>`

El tamaño del array N representado por el atributo 'capacity' y el número de elementos n representado por el atributo 'size'. El método 'size' devuelve el valor del atributo 'size'. Se usa un array de Java (que no son alargables). Se alarga el TAD a $2 \cdot N$ cuando el array está lleno mediante copiado a nuevo array, renombrado (y recogida de basura). [Código, líneas 46-52].

Creación de arrays

[Código, líneas 16 y 48]. Conceptos Java involucrados:

1. No se pueden crear arrays genéricos debido a la implementación de genéricos mediante 'type erasure' [Libro, p90] [\[JP'95, p90\]](#).
2. Variable polimórfica y downcasting.
3. El compilador da una advertencia ('warning').

ERRORES: No es la forma correcta de crear arrays genéricos. Se deben usar los métodos '`getClass`' y '`Array.newInstance`' del paquete de reflexión '`java.lang.reflect.*`':

- [Array.newInstance](#)

Ver código demostrativo en carpeta Generic_array_creation.

Los métodos 'add' y 'remove' pueden tener que mover elementos a derecha e izquierda. [Código, líneas 53 y 63]. Ambos bucles siguen el mismo patrón.

```
#+BEGIN_EXAMPLE
for ( int i = START ; COND(i,END) ; INC(i) )
    SWAP
#+END_EXAMPLE
```

Obsérvese la dualidad en los dos bucles entre los valores START, END, el predicado de condición COND, la función de incremento y el intercambio de valores SWAP. Los valores centinela y SWAP se intercambian, la condición se niega, y el incremento es la inversa.

Ejercicio: ¿Son 'add' y 'remove' métodos duales? En otras palabras, suponiendo que $A.equals(B)$:

- ¿Es cierto $A.add(i,e).remove(i).equals(B)$?
- ¿Es cierto $A.add(i,A.remove(i)).equals(B)$?

Ejercicio: Definir 'add' y 'remove' en términos de un método que mueve, apartir de una posición dada tomada como primer parámetro, los elementos del array a derecha o izquierda según el valor de un segundo parámetro booleano.

Se usa método protegido (protected) 'checkIndex' para comprobar que los índices están en rango y lanzar 'IndexOutOfBoundsException' en caso contrario. Nótese que $r > n-1$ es lo mismo que $r \geq n$.

Complejidad y costes amortizados

Introducción informal a la medida de complejidad $O()$ que veremos en detalle en la siguiente clase. Coste amortizado: coste medio a lo largo de la ejecución. [Transparencias 9-11. Libro p241-242]

Interfaz java.util.ArrayList de la JCF

Lo veremos en la próxima clase junto con otros TADs de la JCF. Mencionamos simplemente que tiene más métodos útiles: 'clear()', 'toArray()', 'indexOf(e)', 'lastIndexOf(e)'.

6.2 Node Lists

Material

- Libro: Sección 6.2
- Transparencias: lists.pdf
- Código:
 - Interfaces:
 - Position.java.
 - PositionList.java.
 - Clases:

- DNode.java.
- NodePositionList.java.
- ElementIterator.java.

Repaso del concepto de lista simple y doblemente enlazada

Repasar [Libro, Capítulo 3, Secciones 3.1 a 3.4], donde se presentan listas simple y doblemente enlazadas, así como algunos ejemplos de uso. En la clase de hoy sólo vemos las listas del Capítulo 6 que son más abstractas. Las listas enlazadas del Capítulo 3 no usan genéricos y manipulan los nodos internos de las listas directamente [Libro, p127]:

```
#+BEGIN_EXAMPLE
public class DList { //
    ...
    public DNode getPrev(DNode v) {...}
    ...
}
#+END_EXAMPLE
```

Interfaz Position<E>

El acceso a los elementos de IndexList se hace usando índices. En las listas de los Capítulo 3 se hace usando nodos.

Interfaz Position<E>:

- Tiene un único método que devuelve el elemento de tipo 'E' asociado a esa posición.
- Abstrae la noción de nodo, oculta detalle de implementación. Los usuarios de la lista no tienen que saber nada sobre la implementación de los nodos. Comparar:
 - [Libro, Capítulo 3, p127]: 'public DNode getPrev(DNode v)'
 - [Libro, Capítulo 6, p247]: 'public Position<E> prev(Position<E> p)'

Además de que se usa un parámetro genérico 'E' para el tipo de los elementos, una posición encapsula un nodo (implementación), del cual sólo interesa acceder al elemento almacenado. Todos los métodos del interfaz trabajan con posiciones. Los nodos reales **implementarán** el interfaz Position<E>.

- Una posición se concibe de forma **relativa**, en función de la posición anterior ('before') y la posición posterior ('after').

Clase DNode<E>

Implementa Position<E> y representa un nodo para una lista doblemente enlazada. A resaltar: el constructor, los "getters" y "setters", método 'element' e 'InvalidPositionException'.

Ejercicio: ¿Podríamos hacer más comprobaciones 'InvalidPositionException'? ¿Podría ser DNode<E> una clase anidada de NodePositionList<E>? ¿Qué modificador de visibilidad tendría que tener una DNode<E> anidada dentro de NodePositionList<E>?.

Ejercicio: ¿Podríamos usar los nodos de una lista doblemente enlazada para implementar una lista simplemente enlazada?

Interfaz `PositionList<E>`

Extiende `Iterable<E>`. A resaltar: métodos `'remove'`, `'positions'`, e `'iterator'`. Los métodos `'first'` y `'last'` no lanzan excepciones.

Ejercicio: *¿Tiene sentido un método `'insertAt(Position<E> p, E e)'`?*

Clase `NodePositionList<E>`

Implementa `PositionList<E>` mediante una lista doblemente enlazada. Nodos especiales `'header'` y `'trailer'` que no almacenan elementos. Cuándo una posición no es válida [Libro, p245]:

- Vacía (null)
- Previamente borrada de la lista.
- Es una posición de otra lista.
- La operación la hace inválida, p.ej, `'L.prev(L.first())'`

Comparamos con el código del método `'checkPosition'`. (Nótese además el downcasting en [Código, línea 37] y la cláusula `'catch'`.)

Los métodos `'first'` y `'last'` lanzan `'EmptyListException'` (lo cual es correcto) aunque el interfaz no lo declara. La excepción está declarada en el fichero `EmptyListException.java`.

Iterables e iteradores: el método `'positions'` crea una nueva `NodePositionList` con las posiciones de la lista y el método `'iterator'` crea un `ElementIterator<E>`. (De momento no entramos en los detalles de estos dos métodos. Volveremos a ellos cuando veamos [6.3 Iterators](#) y [Ejemplos de iteradores](#).)

Ejercicio: *¿Por qué `'first'` y `'last'` no declaran `'EmptyListException'`? (Pista: buscar en un manual de Java la diferencia entre excepciones `'checked'` y `'unchecked'`). ¿Deberían hacerlo?*

Listas: comentarios y ejercicios

¿Cuál es el criterio para elegir entre los TADs lista?

Respuesta: modo de uso de los métodos y complejidad requerida.

Tradicionalmente los TADs (incluidos las listas) se usan para insertar, buscar y borrar **elementos** directamente o a través de índices, no se usan abstracciones como `Position<E>`. Ver [6.3.4 List Iterators in Java](#). Sin embargo, `Position<E>` es un patrón de diseño atractivo y útil que además permite obtener una buena complejidad a los métodos de una implementación de `PositionList<E>`. Los TADs pueden entenderse en función de su implementación (como TADs **para** la implementación de otros TADs) y en función de su propósito. El uso de `Position<E>` encaja en el primer apartado, como veremos más adelante con los árboles. Ver [7.1 General Trees](#).

Pensad en los métodos de inserción y borrado:

- put(e), remove(e) : 'e' elemento, ¿la posición está fija?.
- put(i,e), remove(i) : 'i' índice.
- put(p,e), remove(p) : 'p' posición dada por una clase.

Ejercicio: *¿Abstrae una posición la noción de índice? Implementar `IndexList<E>` usando `NodePositionList<E>` como representación.*

Ejercicio: *Dado un elemento, si no se conoce su posición hay que buscarla. Hay un ejemplo de uso de `PositionList<E>` en [Libro, 6.5 Case Study: The Move-to-Front Heuristic]. Se usa una función 'find(e)' para encontrar la posición de un elemento. ¿Qué complejidad tiene?*

Ejercicio: Comparar el siguiente interfaz de lista:

```

#+BEGIN_EXAMPLE
interface List<E> extends Iterable<E> {
    public boolean isEmpty() ;
    public Integer length() ;
    public E head() throws EmptyListException ;
    public List<E> tail() throws EmptyListException ;
    public E atIndex(Integer i) throws BoundaryViolationException ;
    public List<E> append(List<E> l, List<E> r) ;
    public Iterable<E> take(Integer i) ;
    public Iterable<E> drop(Integer i) ;
    public Iterable<E> elements() ;
    public Iterator<E> iterator() ;
}
#+END_EXAMPLE

```

Con los interfaces `IndexList<E>` y `PositionList<E>` y discutir sobre las ventajas e inconvenientes de cada uno así como posibles implementaciones de unos en términos de otros. Defínase una clase que implemente este interfaz y que tenga un constructor que construye una lista vacía y un método que permite insertar un elemento al principio o al final de la lista.

Viernes 25-02-2011

6.3 Iterators

Material

- Libro: Sección 6.3
- Transparencias: iterators.pdf
- Código: ElementIterator.java, PositionList.java, NodePositionList.java.

Concepto de Iterador

Patrón de diseño ('design pattern'): permite el recorrido **lineal** de los elementos de un TAD (que contiene elementos, no todos los TADs son colecciones de elementos) abstrayendo la implementación de éste.

El iterador guarda (una referencia a un objeto) elemento del TAD, que llamamos **cursor**.

[Transparencia 2]: "Extends the concept of position". [Libro, p254]: "Thus, an iterator extends the concept of the position ADT we introduced in Section 6.2. In fact, a position can be thought of as an iterator that doesn't go anywhere". Ver [Interfaz Position<E>](#). Puede verse Position<E> como un iterador sobre una colección que contiene un único elemento.

Interfaces Iterator<T> e Iterable<T>

Declaraciones

#+BEGIN_EXAMPLE

```
public interface Iterator<E> {
    public E next() ;           /* obligatorio implementarlo */
    public boolean hasNext() ;  /* obligatorio implementarlo */
    public void remove() ;      /* ¡cuidado! */
}

public interface Iterable<E> {
    Iterator<E> iterator() ;
}
```

#+END_EXAMPLE

Cómo iterar sobre TADs

Hacer que el interfaz del TAD extienda Iterable<E>:

#+BEGIN_EXAMPLE

```
public interface TAD<E> extends Iterable<E> {
    ... /* métodos del TAD */
    public Iterator<E> iterator() ;    /* para iterar sobre TAD<E>. */
    public Iterable<E> elements() ;    /* para iterar sobre otro TAD */
}
```

/* con los mismos elementos. */

}

#+END_EXAMPLE

El método 'iterator' se hereda de Iterable<E>. El método 'elements' se define como parte de TAD<E>, no se hereda de Iterable<E>. No es obligatorio definirlo. Es muy útil: 'snapshot' [Libro, p257]: persistencia y mutabilidad (borrado, [JP'05, p102]).

Implementar el TAD e implementar el iterador:

#+BEGIN_EXAMPLE

```
public class TADImplement<E> extends TAD<E> {
    public Iterator<E> iterator() { return new TADIterator<E>(this); }
    ...
}

public class TADIterator<E> extends Iterator {
    E cursor;

    public TADIterator(TAD<E> t) { /* código aquí */ }
    public boolean hasNext()      { /* código aquí */ }
    public E next()                { /* código aquí */ }
    public void remove()          { /* código aquí */ }
}
```

#+END_EXAMPLE

Siempre que sea posible, TADIterator<E> debe iterar sobre TADImplement<E> utilizando los métodos del interfaz TAD<E> y no accediendo a los atributos de TADImplement<E>. Así podremos cambiar el código TADImplement<E> sin tener que cambiar el código de TADIterator<E>.

Otra forma de poder iterar sobre un TAD (utilizada también en el código del libro) es importar 'java.util.Iterator' y definir el método 'iterator' como método propio de TAD<E>:

#+BEGIN_EXAMPLE

```
import java.util.Iterator;

public interface TAD<E> {
    public Iterator<E> iterator() ;    /* para iterar sobre TAD<E>. */
}
```

#+END_EXAMPLE

Pero $\text{TAD}\langle E \rangle$ no es $\text{Iterable}\langle E \rangle$. Entre otras cosas, no se puede usar `for-each` para $\text{TAD}\langle E \rangle$.

También puede definirse `'iterator'` como método de $\text{TADImplement}\langle E \rangle$ y no de $\text{TAD}\langle E \rangle$. El código del libro hace esto con algunas clases que veremos más adelante.

Ejercicio: *¿Qué implicaciones tiene lo descrito en el punto anterior?*

Ejercicio: *¿Por qué hay que importar `java.util.iterator` si el método `'iterator'` se define en el interfaz o en la clase?*

Significado

Los nombres de los métodos de $\text{Iterator}\langle E \rangle$ no son del todo afortunados. Detallamos su significado:

- `hasNext()` : Si el cursor referencia un elemento.
- `next()` : Devuelve el elemento al que referencia el cursor y avanza el cursor.

Hay que preguntar con `'hasNext'` (p.ej, en un condicional) antes de invocar `'next'`.

El cursor referencia null cuando el TAD está vacío o cuando `'next'` devuelve el último elemento de la iteración (no puede avanzar más el cursor).

Ejemplo: Sea $\{A,B\}$ un TAD iterable con 2 elementos:

- Inicialmente el cursor referencia al primer elemento porque el TAD no es vacío.

$\{A,B\}$
↑
cursor

`hasNext()` devuelve `true`.

`next()` devuelve `A` y avanza el cursor a `B`.

- $\{A,B\}$
↑
cursor

`hasNext()` devuelve `true`.

`next()` devuelve `B` y avanza el cursor a `null`.

- $\{A,B\}$

cursor -> null

`hasNext()` devuelve `false`.

`next()` lanza `NoSuchElementException`.

- Si el TAD está vacío entonces el cursor es `null`.

hasNext() devuelve false.

next() lanza NoSuchElementException.

El método 'remove' del iterador no se describe en detalle en [Libro, p. 254, 'Simple Iterators in Java']. No es obligatorio implementar la funcionalidad completa.

- **Incompleto:** sólo lanza 'UnsupportedOperationException' (cuando es invocado). Este es el procedimiento habitual en la JCF para métodos incompletos invocados.
- **Completo:** debe borrar del TAD el elemento al que referencia el cursor, lanzando 'IllegalStateException' si el cursor es null.
- Al iterar un TAD, éste sólo puede modificarse (borrar elementos) con el 'remove' **del iterador**. No se deben usar métodos de inserción o borrado del TAD durante la iteración pues el resultado de la misma puede ser impredecible.

Cómo iterar sobre TADs con bucles 'while' y 'for'

Bucles WHILE:

```
#+BEGIN_EXAMPLE

    Iterator<E> it = TAD<E>.iterator();

    while(it.hasNext()) {

        /* se hace algo con it.next() */

    }

#+END_EXAMPLE
```

Bucles FOR:

```
#+BEGIN_EXAMPLE

    for (Iterator<E> it = TAD<E>.iterator(); it.hasNext(); ) {

        /* se hace algo con it.next() */

    }

#+END_EXAMPLE
```

Se pueden declarar varios iteradores simultáneamente sobre un mismo TAD:

```
#+BEGIN_EXAMPLE

    E e;

    for (Iterator<E> it1 = TAD<E>.iterator(); it1.hasNext(); ) {

        e = it1.next();

        for (Iterator<E> it2 = TAD<E>.iterator(); it2.hasNext();

it2.next())
```

```

        System.out.print(e);
    }

    ##+END_EXAMPLE

```

Ejercicio: *¿Cuál sería en general la salida del código anterior?*

Bucle 'for-each' de Java

[Libro, p. 256], [JP'05, p. 50]

- [Statements](#)

Patrón de sintaxis:

```

    ##+BEGIN_EXAMPLE

        for (type elem : expr) { stmts }

    ##+END_EXAMPLE

```

Donde:

- 'elem' tiene tipo 'type' y no ocurre en 'expr'.
- 'expr' tiene tipo `Iterable<T>` o tipo array de T, con T un subtipo o un boxing de 'type'. Se evalúa 'expr' obteniéndose un objeto iterable, se obtiene su iterador y se itera, con 'elem' referenciando el elemento devuelto por el 'next' del iterador en cada paso.

Semántica informal:

1. **Cuando 'expr' es de tipo `Iterable<T>`** el bucle se lee como "ejecuta 'stmt' **para todo** elemento 'elem' del iterable 'expr'". Se trata de una abreviatura de:

```

    ##+BEGIN_EXAMPLE

    for (Iterator<type> it = expr.iterator(); it.hasNext(); ) {

        type elem = it.next();

        stmts

    }

    ##+END_EXAMPLE

```

donde 'it' es una nueva variable que no ocurre en 'stmts'. (Nótese que 'elem' se declara después de usarse 'expr' y por eso 'elem' no puede ocurrir en 'expr'.)

La variable 'elem' puede declararse localmente dentro del bloque del bucle 'for': "A local variable declaration can also appear in the header of a for statement (§14.14). In this case it is executed in the same manner as if it were part of a local variable declaration statement" (Ver [Statements 14.4](#))

Dada la equivalencia entre 'for' y 'while':

```

    ##+BEGIN_EXAMPLE

```

```

        Iterator<type> it = expr.iterator();

        type elem;

        while(it.hasNext()) {

            elem = it.next();

            stmts

        }

#+END_EXAMPLE

```

Aquí 'elem' se declara antes del bloque del 'while'. Podría no hacerse si el compilador optimiza ('variable hoisting') pero entonces el código no es portable.

2. Cuando 'expr' es de tipo array de T se trata de una abreviatura de:

```

#+BEGIN_EXAMPLE

        for (int j = 0; j < expr.length; j++) {

            type elem = v[j];

            stmts

        }

#+END_EXAMPLE

```

Donde 'j' es una nueva variable que no ocurre en 'stmts'.

Ejemplo:

```

#+BEGIN_EXAMPLE

        public int sumaArray(int [] v) {

            int suma = 0;

            for (int i : v)

                suma += i;

            return suma;

        }

#+END_EXAMPLE

```

Dentro de 'stmt' no se tiene acceso al iterador (a una variable que referencie el objeto iterador), sólo al elemento 'elem'. No se pueden invocar 'next', 'hasNext', ni 'remove'.

Esto significa que en principio for-each se usa para iterar sobre **todos** los elementos (de ahí su nombre). Para iterar sólo sobre algunos elementos (por ejemplo, aquellos que cumplan una condición) debe usarse el iterador directamente. Ver [Ejemplos de iteradores](#).

Ejemplos de iteradores

Mostrar los 'n' primeros elementos de un array de caracteres 'v':

- Usando for-each:

```
#+BEGIN_EXAMPLE

for (char c : v)

if (n>0) {

    System.out.print(c + " ");

    n--;

}

#+END_EXAMPLE
```

Se recorre todo el array por lo que sólo debe usarse para arrays pequeños. Ver [4.2 Analysis of Algorithms](#).

Solución con 'break' (error de estilo):

```
#+BEGIN_EXAMPLE

for (char c : v)

if (n == 0) break;

if (n>0) {

    System.out.print(c + " ");

    n--;

}

#+END_EXAMPLE
```

Un principio de programación estructurada [Bruce J. MacLennan, "Principles of Programming Languages: Design, Evaluation and Implementation"]:

"Structure: The static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations."

Corolario para bucles: Debe salirse fuera de un bucle únicamente cuando la condición sea falsa, teniéndose así un único punto de salida a tener en cuenta en el diseño y demostración de propiedades (p.ej. invariantes) del bucle. No deben usarse 'break', 'continue' o 'return' para salir de un bucle. Desafortunadamente, el código del libro no sigue este principio. La solución idónea a este problema se muestra en el siguiente punto.

- Los arrays no tienen iteradores:

```
#+BEGIN_EXAMPLE

for (Iterator<char> it = v.iterator(); n>0 && it.hasNext(); n--)
```

```
System.out.print(it.next() + " ");
```

```
#+END_EXAMPLE
```

Este código tiene dos errores:

1. Los tipos base ('char') no se pueden pasar como parámetros genéricos, tendría que usarse 'Character' y autoboxing.
2. Los arrays no extienden Iterable<T> y por tanto no tienen un método 'iterator'.

Hay que usar índices (Solución idónea):

```
#+BEGIN_EXAMPLE
```

```
for (int i = 0; n > 0 && i < v.length(); i++, n--)
```

```
System.out.print(v[i] + " ");
```

```
#+END_EXAMPLE
```

Ejemplos en PositionList.java y NodePositionList.java

- ElementIterator<E> implementa Iterator<E> para PositionList<E>.
- PositionList<E> extiende Iterable<E>. El método 'positions' es lo que hemos llamado 'elements' en [Cómo iterar sobre TADs](#).
- NodePositionList<E> implementa el método 'iterator' [Código, línea 160] y usa el iterador en los métodos 'toString' [Código, línea 216] y 'forEachToString' [Código, línea 202].
- Obsérvese que en el bucle for-each de éste último se lleva cuenta del número de elementos ya iterados en una variable auxiliar para poder determinar cuándo concatenar la coma, ya que no puede invocarse 'hasNext' dentro del bucle. Ver [Bucle 'for-each' de Java](#).

Más ejemplos en [9.1.2 A Simple List-Based Map Implementation](#). En general, los TADs iterables deben recorrerse usando iteradores, que abstraen todo lo relativo al posicionado del cursor en el TAD. Usar directamente los métodos del TAD para moverse puede ser más complicado.

Ejercicio: Definir el interfaz `IterableIndexList<E>` que extiende `IndexList<E>` con el método 'iterator'. Implementar la clase `IterableArrayIndexList<E>` que implemente `IterableIndexList<E>` y además implemente los métodos estáticos 'toString' y 'forEachToString' similares a los de `PositionList<E>`.

6.3.4 List Iterators in Java

- [Transparencia 6, iterators.pdf]
- [Libro, Sección 6.3.4 'List Iterators in Java']

Ver también:

- [Listas: comentarios y ejercicios](#)
- [6.4.2 Sequences](#)
- [6.4 List ADTs and the Collections Framework](#)

Las listas extienden `Collection<E>` que a su vez extiende `Iterable<E>`.

No existe `Position<E>` en JCF, se usan índices y elementos.

Interfaces: `List` (usa índices) y `ListIterator`. Hay además clases abstractas: `AbstractList`, `AbstractSequentialList`, etc.

Implementaciones (Java Platform SE 6): `ArrayList`, `LinkedList`, `AttributeList`, `CopyOnWriteArrayList`, `RoleList`, `RoleUnresolvedList`, `Stack`, `Vector`.

'Fail-fast' para múltiple iteradores cuando uno modifica la colección [Libro, p259].

6.4.2 Sequences

- [Transparencias 7-8, iterators.pdf], [Libro, p264].
- Código: `Sequence.java` (sólo interfaz).
El interfaz `Sequence<E>` es la unión de los interfaces `PositionList<E>`, `IndexList<E>` y `Deque<E>`. Este último no lo hemos visto en clase. Se describe en [Libro, Sección 5.3, 'Double-Ended Queues'].
- ERRORES: En [Transparencia 7] sólo se dice que las secuencias son la unión de Array Lists y Node Lists.

Métodos 'atIndex' e 'indexOf': puente entre posiciones e índices.

Se recomienda estudiar esta sección del libro para abordar los ejercicios de [Listas: comentarios y ejercicios](#).

4.2 Analysis of Algorithms

Material

- Libro: Sección 4.1 y Sección 4.2
La sección 4.1 repasa preliminares matemáticos: funciones constante, logaritmo, lineal, n -log- n , cuadrática, cúbica, polinomial y exponencial. Sumas aritméticas y geométricas.
Razones de crecimiento. Funciones techo ('ceiling') y suelo ('floor').
- [Libro, Capítulo 4, Sección 4.2, 'Analysis of Algorithms']
- Transparencias: `analysis.pdf`

Resumen de principios fundacionales

Complejidad, eficiencia, coste de un algoritmo: medida del tiempo de ejecución (tiempo) y del uso de memoria (espacio).

El pseudocódigo es una notación que abstrae (alto nivel) las particularidades de los lenguajes de programación (de un paradigma de computación).

La descripción de algoritmos mediante pseudocódigo permite su análisis teórico independiente de lenguajes y hardware concretos.

La complejidad de las operaciones primitivas de un algoritmo descrito en pseudocódigo es proporcional (función constante) a la complejidad real de su implementación en un lenguaje de programación concreto ejecutándose en un hardware concreto. Por tanto, la complejidad del algoritmo en pseudocódigo es proporcional (función constante) a la complejidad de su implementación. Dicho de otro modo, los factores a tener en cuenta en una implementación real afectan la complejidad teórica en un factor constante

[Transparencia 14]. Más correctamente, afectan la complejidad sin relevancia asintótica. Ver [Notación O\(\)](#).

La complejidad de un algoritmo incrementa con el incremento del tamaño de la entrada ('input size'). La complejidad es una **función** de dicho tamaño.

Análisis experimental: requiere análisis estadístico (con buen muestreo de datos de entrada) y probabilístico (distribución de dichos datos). Independientemente de dicha distribución, se pueden identificar casos mejores y peores.

El estudio del caso peor nos da unos resultados más precisos y con menor variabilidad que los estudios de caso medio y los experimentales.

Los algoritmos eficientes en el caso peor lo serán también en los otros casos.

Notación O()

Intuición: establecer la complejidad de una función del tamaño de la entrada indicando otra función proporcional que la acota asintóticamente.

Definición Matemática:

Sean f y g dos funciones de naturales a reales. Se dice que $f(n)$ **es del orden de** $g(n)$, o $f(n)$ es $O(g(n))$, si existe una constante real $c > 0$ y una constante natural $n_0 \geq 1$ tal que:

$$f(n) \leq c \cdot g(n) \text{ para } n \geq n_0$$

Medida asintótica: n tiende a infinito.

Definición en Inglés: [Transparencias 21 y 30].

Análisis de complejidad: se calcula $f(n)$ y se determina el "mínimo" $g(n)$ [Transparencias 20-23] [Libro, p173, 'Characterizing Functions in Simplest Terms'].

Se ignoran factores constantes y de orden inferior [Transparencias 20-23].

Pero, **CAUTION**: [Libro, p176, 'Some Words of Caution']: "we should at least be somewhat mindful of the constant factors and lower order terms we are "hiding"". Ejemplo: $O(10^{100} \cdot n)$.

Escala típica de complejidad (de menor a mayor):

1. Constante: $O(1)$
2. Logarítmica: $O(\log n)$

3. Lineal: $O(n)$
4. N-Log-N: $O(n \log n)$
5. Cuadrática $O(n^2)$
6. Cúbica: $O(n^3)$
7. Polinomial: $O(n^m)$
8. Exponencial: $O(2^n) \dots O(m^n)$

Notaciones Omega y Theta

[Transparencias 29-31] [Libro, p174]

Complejidad: comentarios y ejercicios

Ejercicio: ¿Es realista que el coste de la evaluación de expresiones y del indexado en arrays sea constante? [Transparencia 11]. (El indexado de un array se hace en base a expresiones y es asimismo una expresión.) Comparar [Transparencia 11] con la descripción de 'Primitive Operations' en [Libro, Sección 4.2.2] y señalar las diferencias notables. ¿Tendría sentido afirmar que la igualdad (==) de expresiones tiene complejidad constante? ¿Cómo se implementa la igualdad de objetos en Java?

La invocación de métodos no tiene complejidad constante, depende de la complejidad del método [Libro, Sección 4.2.3, p170]: "except for method calls, of course".

Preguntarse si el análisis de algoritmos descritos en pseudocódigo es realmente un análisis teórico de alto nivel. ¿Semántica del pseudocódigo? ¿Atado a un paradigma de programación o modelo de computación RAM Model [Transparencia 8]? ¿Se hacen suposiciones realistas sobre el coste de la recursión? ¿Se ignora el potencial para el paralelismo? ¿Es realista ignorar que un algoritmo ejecutará en un programa dentro de un entorno junto con otros programas bajo el control de un S.O. y bajo las restricciones del hardware? ¿Se necesita en la práctica tanto el análisis teórico como el experimental?

"In theory there is no difference between theory and practice. In practice is." Yogi Berra.

6.4 List ADTs and the Collections Framework

Las listas de la Java Collections Framework (JCF) ya las hemos visto por encima en [6.3.4 List Iterators in Java](#).

Interfaces importantes: `Collection<E>` y `Map<K,V>`.

Diagramas de clases:

- Con genéricos (JDK 1.6):

- [Java-collections-cheatsheet-v2](#)
- [Java 1.6 - collections class](#)

- Sin genéricos (JDK 1.4, se usa 'Object' en vez de 'E'). Damos las URLs porque los diagramas de clase están bien organizados:

- [Java util Collection](#)
- [Java util Map](#)

A resaltar:

- Collection<E> extiende Iterable<E> pero Map<K,V> no.

Ejercicio: *¿Significa esto que una implementación de Map<K,V> no puede ser iterable?*

- Muchos más métodos útiles en las clases: conversión a array y viceversa, reordenación ('sort', 'shuffle', 'reverse', etc), computación con segmentos, búsqueda por patrones, etc.
- Familia de implementaciones típicas, código no disponible.

Viernes 04-03-2011

7 Tree Structures

Material

- Libro: Capítulo 7
- Transparencias: trees.pdf

7.1 General Trees

General Trees: Terminología y Definiciones

Motivación: organizar información de forma jerárquica.

Utilizados en la implementación de otros TADs.

La JCF no incluye una implementación de árboles generales, se utilizan árboles en implementaciones de otros TADs (por ejemplo, Maps). Ver [Listas: comentarios y ejercicios](#).

Definición recursiva [Libro, p281, últimas tres líneas]:

Un **árbol general** es o bien vacío o bien un nodo raíz que contiene un elemento y un conjunto de cero o más (sub)árboles hijos. (Nótese que en un conjunto no hay orden ni repetición).

Definición más formal [CLRS'01, B.5], [AHU'83, p231]:

Un **árbol libre** ('free tree') es un grafo conectado, no-dirigido y acíclico. Un **árbol ordinario** ('rooted tree') es un árbol libre en el que se elige un nodo como raíz (se orientan las aristas al representarlo gráficamente).

Terminología [Transparencia 3]:

- **Raíz** ('root'): nodo sin padre.
- **Nodo interno** ('internal node'): nodo con al menos un hijo.

- **Nodo externo** ('external node'): nodo sin hijos.
- **Subárbol** ('subtree'): nodo considerado como raíz y todos sus descendientes.

ERRORES: La descripción de ancestro y de descendiente en [Transparencia 3] es incompleta.

Terminología que no está o que completa [Transparencia 3] y que está en el libro o en otras fuentes:

- **Ancestro** de un nodo v : un nodo w es ancestro de v si $v=w$ o w es ancestro del padre de v [Libro, p282].
- **Descendiente** de un nodo w (la inversa de ancestro): v es descendiente de w si w es ancestro de v [Libro, p282].
- **Nodo hoja**: nodo externo. Usaremos estos dos nombres indistintamente.
- **Hermano** ('sibling') de un nodo: nodo con el mismo padre.
- **Arista** ('edge') de un árbol: par de nodos en relación padre-hijo o hijo-padre.
- **Grado** ('degree') de un nodo: el número de hijos del nodo. Se puede extender ésta definición a todo el árbol: el **grado de un árbol** es el máximo de los grados de todos sus nodos.
- **Camino** ('path') de un árbol: secuencia de nodos tal que cada nodo consecutivo forma una arista. La **longitud del camino** es el número de aristas.
- **Árbol ordenado** ('ordered tree'): existe un orden lineal (total) definido para los hijos de cada nodo: primer hijo, segundo hijo, etc. Se visualiza dibujando los hijos en orden de izquierda a derecha bajo el padre. Ejemplo: capítulos de un libro.
- **Profundidad** ('depth') y **altura** ('height') de un nodo. [Libro, Sección 7.2.1 'Depth and Height']:
 - o La **profundidad de un nodo** es la longitud del camino desde ese nodo hasta la raíz (o viceversa). La longitud del camino es cero si el nodo es raíz. (El libro la define de forma equivalente como el número de ancestros "propios" del nodo).
 - o La **altura de un nodo** es la longitud del mayor de todos los caminos que hay del nodo a las hojas.
 - o Damos otras definiciones formales (usando código Java) en [7.2 Tree Traversal Algorithms](#).
 - o **Altura de un árbol**: la altura de la raíz.
 - o La **profundidad de un árbol** podría definirse como la mayor de las profundidades de las hojas, pero ese valor es igual a la altura. De hecho, la altura de un árbol también se define como el máximo de la profundidad de todos sus nodos [CLRS'90, p94].
 - o Profundidad y altura se han definido como propiedades de nodos. Por tanto, según estas definiciones **no tiene sentido hablar de la profundidad o la altura de un árbol vacío** (sin nodos).
- **Nivel** ('level'): conjunto de nodos con la misma profundidad. Así, tenemos desde el nivel 0 hasta el nivel 'h' donde 'h' es la altura del árbol.

Interfaz Tree<E>

Código: Tree.java, InvalidPositionException, EmptyTreeException.java, BoundaryViolationException.java.

Interfaz de árboles generales. Como en las listas enlazadas (Ver [6.2 Node Lists](#)) se usa el [Interfaz Position<E>](#) para abstraer la implementación de los nodos de un árbol. De nuevo, una posición se entiende de forma relativa en función de las posiciones vecinas (que serán los padres y los hijos).

En el libro, el código y las transparencias se usa la palabra "nodo" para referirse a una "posición" (interfaz) no a la implementación. Nosotros también seguimos esta convención.

Es un interfaz pensado para trabajar directamente con posiciones (abstracciones de nodos). Los TADs de árboles se suelen usar en implementaciones de otros TADs y para ello se necesita poder trabajar eficientemente con nodos.

Esto explica que no es un interfaz recursivo, cuando sí lo es la definición de árbol general. Los métodos trabajan con posiciones y no con árboles. Por ejemplo, se define:

```

#+BEGIN_EXAMPLE

public Iterable<Position<E>> children(Position<E> v) throws ...

#+END_EXAMPLE

```

y no:

```

#+BEGIN_EXAMPLE

public Iterable<Tree<E>> children(Tree<E> t) throws ...

#+END_EXAMPLE

```

Por esta razón, los métodos de las clases que **usen** el interfaz tomarán siempre un árbol como argumento para poder invocar los métodos del interfaz. Por ejemplo, [7.2 Tree Traversal Algorithms](#). El interfaz importa 'java.util.Iterator' y define el método 'iterator' directamente, no extiende Iterable<E>. Esto significa que un Tree<E> no es un Iterable<E>, lo cual tiene sentido por lo que se ha dicho arriba sobre que se trabaja directamente con posiciones. Ver [Cómo iterar sobre TADs](#).

El método 'iterator' devuelve un iterador para iterar sobre todos los nodos del árbol.

Hay dos métodos que devuelven Iterable<E>:

1. Método 'positions', que es lo que hemos llamado 'elements' en [Cómo iterar sobre TADs](#).
2. Método 'children', que devuelve un iterable con los hijos del nodo. Si el árbol está ordenado el iterable estará ordenado según el orden de los hijos.

Ejercicio: ¿Si el árbol está ordenado debe 'iterator' recorrer el árbol acorde al orden de nodos en el árbol?

Algunos métodos lanzan excepciones cuando las posiciones no son válidas:

- 'BoundaryViolationException' lanzada por 'parent' cuando el nodo argumento es la raíz.
- 'EmptyTreeException' lanzada por 'root' cuando el árbol es vacío (definida en EmptyTreeException.java).
- 'InvalidPositionException' lanzada por métodos que toman posiciones como argumento cuando la posición es espúrea.

Sólo se define un método modificador 'replace' para modificar el elemento de una posición del árbol. [Libro, p284, últimas líneas]: "we prefer to describe different tree update methods in conjunction with specific applications of trees in subsequent chapters. In fact, we can imagine several kinds of tree update operations beyond those given in this book".

Interfaces y clases de la implementación de Tree<E>

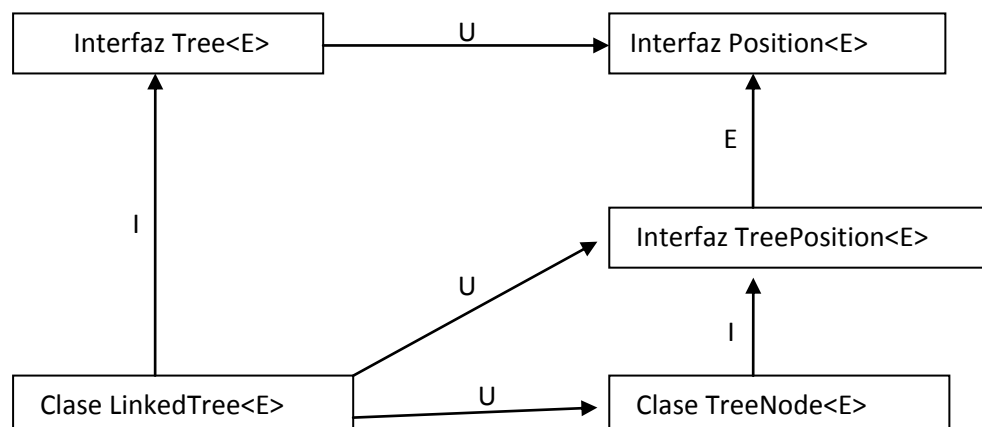
Cuestiones de implementación de árboles generales

Representación de nodos: [Transparencia 16, trees.pdf] [Libro, p286, 'A Linked Structure for General Trees'].

- Un nodo tiene referencias al padre, al elemento, y a una colección que contiene los nodos hijos.
- En general se sigue la siguiente convención en las figuras de árboles del libro y las transparencias: se dibujan los nodos internos como círculos y los externos como cuadrados.

El libro no muestra ni explica las interfaces y clases utilizadas en la implementación de Tree<E>. Se centrará más en implementaciones de árboles binarios y derivados. Tampoco ofrecerá código para TADs que utilizarían el interfaz de árboles generales, por ejemplo, [10.4 \(2,4\) Trees](#).

Diagrama de clases



U: Usa. I: Implementa. E: Extiende.

Interfaz `TreePosition<E>`

- **Código:** `TreePosition.java`.

Errores: Los comentarios iniciales son incorrectos. No es un interfaz para nodos 'binary tree' sino para nodos de árboles generales.

Interfaz intermedio que define "setters" y "getters" para los atributos (padre, elemento, colección de hijos) que tendrá la implementación de un nodo. El método 'getChildren' devuelve, y el método 'setChildren' toma, un `PositionList` de nodos. Sus implementaciones devolverán y tomarán respectivamente objetos que implementen el [Interfaz `PositionList<E>`](#).

Esto no significa que la la clase que implemente los nodos (la clase que implemente `TreePosition<E>`) tenga necesariamente que utilizar una implementación de `PositionList` internamente como colección para almacenar los nodos. Sin embargo, así será [Clase `TreeNode<E>`](#).

Ejercicio: ¿Sería más adecuado utilizar `Collection<Position<E>>` o `Iterable<Position<E>>` para 'getChildren' y 'setChildren'?

Clase `TreeNode<E>`

- **Código:** `TreeNode.java`.

Implementa `TreePosition<E>`.

Errores: Los comentarios iniciales son incorrectos, no es una clase para implementar nodos de 'binary tree' sino de árboles generales.

Tres atributos: referencia al nodo padre, referencia al elemento, referencia a la `PositionList` con los nodos hijos.

Dos constructores, uno que crea un nodo vacío (en el que todos los atributos quedan a null) y otro que construye un nodo dados valores para los atributos.

Se implementa el método 'element' del [Interfaz `Position<E>`](#).

ERRORES: Los métodos 'getChildren' y 'setChildren' almacenan y devuelven respectivamente la referencia a la `PositionList`, no realizan una copia de la lista. Esto puede romper la abstracción de datos: la lista puede modificarse desde "fuera" del árbol. Se puede modificar el contenido y/o orden de nodos usando los métodos de la lista directamente sin utilizar los métodos de borrado de las clases que implementen el árbol.

Clase `LinkedTree<E>`

- **Código:** `LinkedTree.java`.

Implementa `Tree<E>`.

Se definen sólo dos atributos: el tamaño 'size' y el nodo raíz 'root'. El constructor crea un árbol vacío: 'root' a null y 'size' a 0. El método protegido 'checkPosition' [Código, línea 118] es

utilizado por varios métodos para comprobar la validez de un nodo (de un objeto que implementa el [Interfaz Position<E>](#)). Un nodo es válido si no es null y es un objeto que implementa el [Interfaz TreePosition<E>](#). Si el nodo es válido se devuelve con casting a 'TreePosition'. Los métodos que invocan 'checkPosition' utilizan una variable temporal ('vv', 'ww') de tipo 'TreePosition' para almacenar el objeto porque el tipo estático ha sido cambiado por el casting. Los métodos 'isInternal' e 'isExternal' implementan exactamente su definición dada en [General Trees: Terminología y Definiciones](#). Obsérvese que 'isExternal' comprueba que la 'PositionList' es null o vacía: hay un constructor de la [Clase TreeNode<E>](#) que construye nodos con todos los atributos a null y por tanto la lista puede ser null. Los métodos 'iterator' y 'positions' devuelven como iterable un objeto de la [Clase NodePositionList<E>](#). El método 'children' devuelve la 'PositionList' almacenada en el nodo, que es obtenida a través del método 'getChildren' de la [Clase TreeNode<E>](#).

Se definen métodos modificadores que no están en el [Interfaz Tree<E>](#): 'addRoot' y 'swapElements'. Recuérdese que en dicho interfaz sólo hay un método modificador: 'replace'.

Se define un método 'createNode' [Código, línea 125] que devuelve un objeto creado por el constructor de 'TreeNode'. El método 'createNode' solamente es invocado por el método modificador 'addRoot'. Este último podría haber invocado directamente el constructor de 'TreeNode'. En esta clase, 'createNode' es innecesario. En la [Clase LinkedBinaryTree<E>](#) que implementa árboles binarios se define un método de igual nombre que sí será usado. Finalmente se define el método 'preordenPositions' que añade los nodos del árbol, recorridos en preorden, al final de una 'PositionList' de nodos que toma como argumento.

7.2 Tree Traversal Algorithms

Material

- [Transparencias 5-6, trees.pdf]
- **Código:** Ejemplos que no son parte del paquete net.datastructures:
 - [ch07-fragments](#)
 - En particular:
 - Trees-depth
 - Trees-height1
 - Trees-height2
 - Trees-preorderPrint
 - Trees-parentheticRepresentation
 - Trees-postorderPrint
 - Trees-diskSpace

Profundidad de un nodo:

```
#+BEGIN_EXAMPLE
```

```
public static <E> int depth(Tree<E> T, Position<E> v) {
```

```

        if (T.isRoot(v))
            return 0;
        else
            return 1 + depth(T, T.parent(v));
    }

    #+END_EXAMPLE

```

Explicamos "static" y "<E>" en [Traversals: comentarios y ejercicio](#).

Altura de un nodo (adaptado de Trees-height2):

```

    #+BEGIN_EXAMPLE

    public static <E> int height(Tree<E> T, Position<E> v) {
        int h = 0;
        if (T.isExternal(v)) return h;
        else
            for (Position<E> w : T.children(v))
                h = Math.max(h, height(T, w));
        return 1 + h;
    }

    #+END_EXAMPLE

```

7.2.2 Preorder Traversal

- [Transparencia 5]
- **Código:** Trees-preorderPrint, Trees-parentheticRepresentation.

```

    #+BEGIN_EXAMPLE

    public static <E> String toStringPreorder(Tree<E> T, Position<E> v) {
        String s = v.element().toString(); // the main "visit" action
        for (Position<E> w : T.children(v))
            s += ", " + toStringPreorder(T, w);
        return s;
    }

    #+END_EXAMPLE

```

7.2.3 Postorder Traversal

- [Transparencia 6]
- **Código:** Trees-postorderPrint, Trees-diskSpace.

```

    #+BEGIN_EXAMPLE

```

```

public static <E> String toStringPostorder(Tree<E> T, Position<E> v) {
    String s = "";
    for (Position<E> w : T.children(v))
        s += toStringPostorder(T, w) + " ";
    s += v.element(); // main "visit" action
    return s;
}

#+END_EXAMPLE

```

Traversals: comentarios y ejercicio

Los métodos que implementan los recorridos toman como parámetros el árbol y un nodo. Hemos explicado las razones en [Interfaz Tree<E>](#).

Los recorridos se implementan mediante métodos genéricos y estáticos. Podemos asumir que son métodos de alguna clase estática no genérica, por ejemplo:

```

#+BEGIN_EXAMPLE

public static class Traversals {
    public static <E> int depth ...
    public static <E> int height ...
    public static <E> String toStringPreorder ...
    public static <E> String toStringPostorder ...
}

#+END_EXAMPLE

```

Ejercicio: Definir dicha clase.

EJERCICIO: ¿Se podría implementar dicha clase de forma que el árbol que se esté recorriendo se almacene en un atributo y no tenga que pasarse como argumento a cada método? En caso negativo, realizar los cambios necesarios en la clase para conseguirlo.

7.3 Binary Trees

Material

- [Trasparencias 7-15, 17-20, trees.pdf]
- Código:
 - Interfaces:
 - BTPosition.java
 - BinaryTree.java
 - Clases:

- BTNode.java
- LinkedBinaryTree.java

Binary Trees: Terminología y Definiciones

[Libro, p296-297]

Caso especial de árbol general en el que todo nodo tiene como máximo 2 hijos, el hijo izquierdo ('left child') y el hijo derecho ('right child'), que están ordenados: el izquierdo precede al derecho.

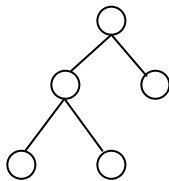
Definición recursiva:

Un árbol binario es un [árbol ordinario](#) que es o bien vacío o bien un nodo raíz con dos (sub)árboles binarios izquierdo y derecho.

Árbol binario propio ('proper binary tree')

También llamado **árbol estrictamente binario** ('strictly binary tree'): todo nodo interno tiene 2 hijos. Equivalentemente, todo nodo tiene o bien 0 o bien 2 hijos. También **2-Tree** o árbol de grado 2.

#+BEGIN_EXAMPLE

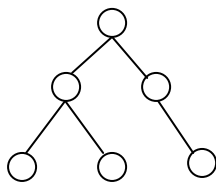


#+END_EXAMPLE

Árbol binario impropio ('improper binary tree')

Árbol binario que no es propio.

#+BEGIN_EXAMPLE

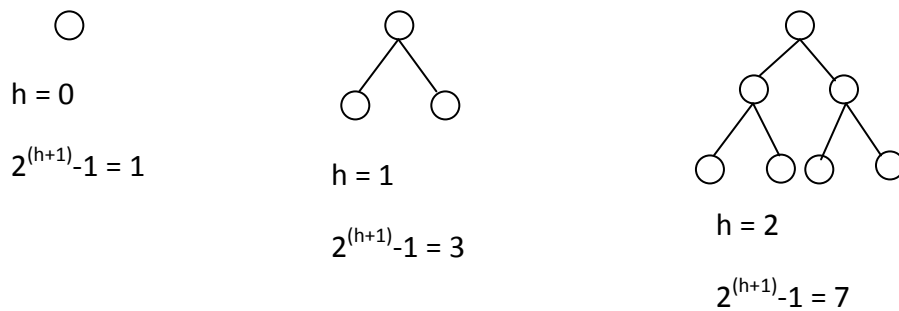


#+END_EXAMPLE

Árbol binario perfecto ('perfect binary tree')

Árbol binario propio con el máximo número de nodos. El número de nodos de un árbol binario (propio o impropio) es como máximo $2^{h+1}-1$. El número de nodos internos de un árbol binario (propio o impropio) es como máximo 2^h-1 . Ver [7.3.3 Properties of Binary Trees](#).

#+BEGIN_EXAMPLE



#+END_EXAMPLE

Obsérvese que todas las hojas están en el mismo nivel y todos los nodos internos tienen dos hijos.

[Libro, p296], [CLRS'01, B.5] definen **árbol binario lleno** ('full binary tree') como sinónimo de árbol propio. Otros ([AHU'83, p106], [Horowitz & Sahni "Data Structures in Pascal"]) lo definen como sinónimo de árbol binario perfecto.

Árbol binario equilibrado ('balanced binary tree')

Para todo nodo, el valor absoluto de la diferencia de altura entre los dos subárboles hijos es como máximo 1. Más formalmente: para todo nodo 'n' con hijo izquierdo 'i' e hijo derecho 'd' se tiene $|h(i) - h(d)| \leq 1$. En otras palabras: para todo nodo con altura 'h', o bien sus dos hijos tienen la misma altura, h-1, o un hijo tiene altura h-1 y el otro h-2. Obsérvese que según la definición, todo subárbol de un árbol equilibrado es equilibrado.

Binary Trees: Algunos Usos

[Transparencias 8-9]

Otros usos: implementación de TADs.

7.3.3 Properties of Binary Trees

[Transparencia 10]

Algunas propiedades se deducen de otras, por ejemplo:

$$\begin{aligned}
 n &= e + i \\
 &\leftrightarrow \{e = i + 1 \leftrightarrow i = e - 1\} \\
 n &= e + (e - 1) \\
 &\leftrightarrow \{\text{aritmética}\} \\
 n &= 2e - 1
 \end{aligned}$$

Otro ejemplo (propiedad que no está en [Transparencia 10]):

$$n = e + i$$

$$\Leftrightarrow \{e = i + 1\}$$

$$n = (i + 1) + i$$

$$\Leftrightarrow \{\text{aritmética}\}$$

$$n = 2i + 1$$

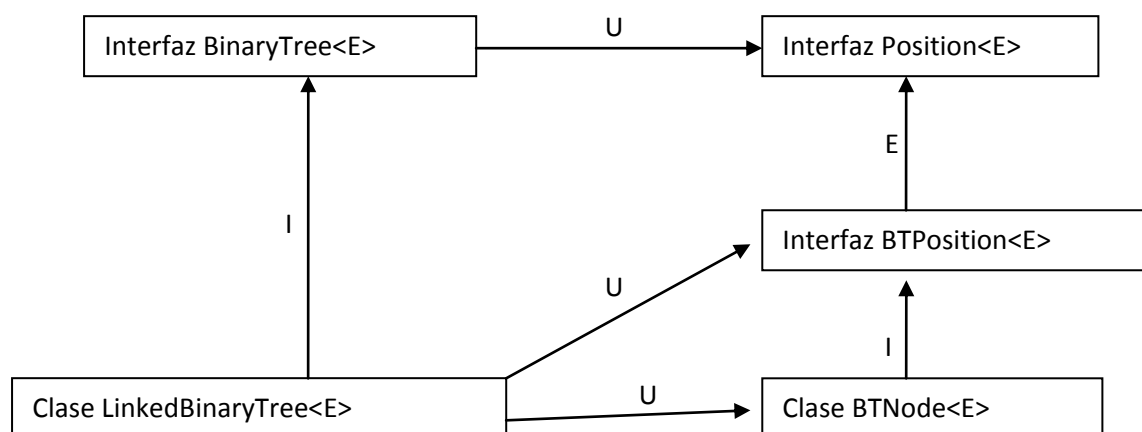
Hay una relación entre $\log(x)$ y 2^x . Recordad: $\log(x) = y \Leftrightarrow 2^y = x$.

Son funciones inversas:

- $\log(2^x) = x$
- $2^{\log(x)} = x$

Interfaces y clases de la implementación de `BinaryTree<E>`

Diagrama de clases



U: Usa. I: Implementa. E: Extiende.

Interfaz `BinaryTree<E>`

- **Código:** `BinaryTree.java`.

Extiende `Tree<E>` con métodos "getters" e interrogadores para nodos hijos que pueden lanzar excepciones. Igual que en [Interfaz `Tree<E>`](#), no se definen métodos modificadores [Libro, p. 298]: "we will consider some possible update methods when we describe specific implementations and applications of binary trees."

Interfaz `BTPosition<E>`

- **Código:** `BTPosition.java`.

Interfaz intermedio con "setters" y "getters" para los atributos (padre, elemento, hijo izquierdo, hijo derecho) que tendrá la implementación de un nodo. Extiende el [Interfaz `Position<E>`](#). No extiende el [Interfaz `TreePosition<E>`](#) usado en los árboles generales.

Clase BTreeNode<E>

- Representación de nodos: [Transparencia 17, trees.pdf] [Libro, p301].
- **Código:** BTreeNode.java.

Implementa BTPosition<E>.

Cuatro atributos: referencia al nodo padre, referencia al elemento, referencia al hijo izquierdo y referencia al hijo derecho. (No se tiene una referencia a un iterable con los hijos como en [Clase TreeNode<E>](#)).

Dos constructores, uno que crea un nodo vacío (en el que todos los atributos quedan a null) y otro que construye un nodo dados valores para los atributos.

Se implementa el método 'element' del [Interfaz Position<E>](#).

Clase LinkedBinaryTree<E>

- [Transparencia 17]
- **Código:** LinkedBinaryTree.java.

Implementa BinaryTree<E>.

Ignoramos de momento los comentarios iniciales del código que explican por qué en esta clase se usa directamente el **atributo** 'size' en vez de invocar el **método** 'size' para obtener el número de nodos del árbol. (Se trata de un caso cuando menos cuestionable de una clase padre cuya implementación está determinada por clases hijas. Volveremos sobre este punto en [10.1 Binary Search Trees](#)).

Se definen sólo dos atributos: el tamaño 'size' y el nodo raíz 'root'. El constructor crea un árbol vacío: 'root' a null y 'size' a 0. El método protegido 'checkPosition' [Código, línea 283] es utilizado por varios métodos para comprobar la validez de un nodo (de un objeto que implementa el [Interfaz Position<E>](#)). Un nodo es válido si no es null y es un objeto que implementa el [Interfaz BTPosition<E>](#). Si el nodo es válido se devuelve con casting a 'BTPosition'.

Los métodos que invocan 'checkPosition' utilizan una variable temporal ('vv', 'ww') de tipo 'BTPosition' para almacenar el objeto porque el tipo estático ha sido cambiado por el casting.

Los métodos 'isInternal' e 'isExternal' implementan exactamente su definición dada en [General Trees: Terminología y Definiciones](#). Los métodos 'iterator', 'positions' y 'children' devuelven como iterable un objeto de la [Clase NodePositionList<E>](#).

Se definen métodos modificadores que no están en el [Interfaz BinaryTree<E>](#):

- Descritos en [Libro, p303]: 'addRoot', 'remove', 'attach', 'insertLeft' e 'insertRight'.
- No descritos en el libro: 'swapElements', 'expandExternal', 'removeAboveExternal'.

Recuérdese que el [Interfaz BinaryTree<E>](#) extiende el [Interfaz Tree<E>](#) el cual sólo declara un método modificador: 'replace'. **Importante:** Estos métodos modificadores se usan junto con el constructor en la [Construcción de árboles binarios](#). Los métodos no descritos en el libro se

usarán en clases que extienden 'LinkedList', como por ejemplo la de los árboles binarios de búsqueda [10.1.3 Java Implementation](#).

Se define un nuevo método observador 'sibling' que devuelve el hermano de un nodo si lo tiene o lanza una excepción si no lo tiene. Se define un método 'createNode' [Código, línea 290] que devuelve un objeto creado por el constructor de 'BTNode'. El método 'createNode' es invocado por los métodos modificadores 'addRoot', 'insertLeft' e 'insertRight'.

Puede observarse que la construcción de nodos (objetos de clase 'BTNode') en la clase 'LinkedList' solamente se hace invocando 'createNode'. Más adelante ([10.1.3 Java Implementation](#) y [10.2.2 Java Implementation](#)) veremos clases que extienden 'LinkedList' y sobrescriben ('override') 'createNode'. Recuérdese que en Java los constructores de clase no pueden sobrescribirse. Por ello se usa un método ordinario, que puede sobrescribirse, para construir nodos.

Se definen los métodos 'preorderPositions' e 'inorderPositions' que añaden los nodos del árbol, recorridos en preorden y en inorden respectivamente, al final de una 'PositionList' de nodos que toman como argumento.

Viernes 11-03-2011

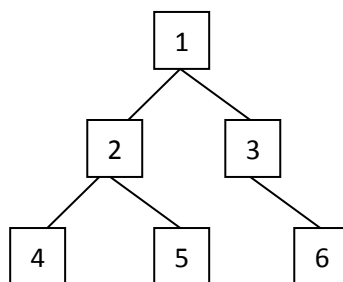
7 Tree Structures (continuación)

7.3 Binary Trees (continuación)

Construcción de árboles binarios

La construcción de un árbol se puede hacer de varias maneras después de llamar al constructor mediante combinaciones de 'addRoot', 'insertLeft', 'insertRight', 'attach', 'expandExternal', etc.

Ejemplo:



Construcción 1:

#+BEGIN_EXAMPLE

```

BinaryTree<Integer> tree = new LinkedList<Integer>();

tree.insertLeft(tree.insertLeft(tree.addRoot(1),2),4);

tree.insertRight(tree.left(tree.root()),5);

tree.insertRight(tree.insertRight(tree.root(),3),6);
  
```

#+END_EXAMPLE

Construcción 2:

```
#+BEGIN_EXAMPLE
```

```
    BinaryTree<Integer> tree = new LinkedBinaryTree<Integer>();

    tree.addRoot(1);

    tree.expandExternal(tree.root(), 2, 3);

    tree.expandExternal(tree.left(tree.root()), 4, 5);

    tree.insertRight(tree.insertRight(tree.root(), 3), 6);
```

```
#+END_EXAMPLE
```

Nótese en ambos casos el uso del **enlazado dinámico**: [Interfaz Tree<E>](#) e [Interfaz BinaryTree<E>](#) no tienen métodos modificadores, éstos los provee la [Clase LinkedBinaryTree<E>](#).

7.3.6 Traversals of Binary Trees

En preorden y en postorden. Se pueden implementar usando los métodos 'left' y 'right', que devuelven respectivamente los nodos raíz de los hijos izquierdo y derecho, en vez usar 'children' y un iterador.

Inorden [Transparencia 12] [Código, LinkedBinaryTree.java, línea 306].

Ejercicio: Implementar 'postorderPositions' (que falta en el código de LinkedBinaryTree.java).

Ejemplos de uso:

- Mostrar y evaluar expresiones aritméticas.
 - [Transparencias 13-14], [Libro, p312-316, junto con otros ejemplos].
- Recorrido Euler: [Transparencia 15 y 19-20], [Libro, p317-322].
 - No lo veremos en clase.

Complejidad de los métodos

- [Libro, página 309]

Para la implementación LinkedBinaryTree.java.

Método	Complejidad
Size, isEmpty	O(1)
Iterator, positions	O(n)
Replace	O(1)
root, parent, children, left, right, sibling	O(1)
hasLeft, hasRight, isInternal, isExternal, isRoot	O(1)
insertLeft, insertRight, attach, remove	O(1)

La complejidad constante se consigue porque los argumentos y valores devueltos por los métodos son nodos del árbol.

8 Priority Queues

Material

- Libro: Capítulo 8.
- Transparencias: priorityqueues.pdf
- Código Libro:
 - Interfaces
 - Entry.java
 - PriorityQueue.java.
 - Clases:
 - DefaultComparator.java
 - EmptyPriorityQueueException.java
 - InvalidKeyException.java
 - SortedListPriorityQueue.java.
- Código Java Platform SE 6:
 - Interfaces:
 - java.util.Comparator<T>
 - java.lang.Comparable<T>.

Motivación de las colas con prioridad

Los TADs que hemos visto hasta ahora se basan en el concepto de **posición** ("position-based data structures" [Libro, p334]). Un elemento está asociado a (contenido en) una posición y las posiciones se organizan de forma relativa, p.ej, en relación lineal anterior-posterior (listas), en relación padre-hijo (árboles).

Las colas con prioridad organizan los elementos acorde a una propiedad de prioridad. La posición que ocupa un elemento dentro de la cola viene determinada por la propiedad de prioridad, por tanto no se necesita el [Interfaz Position<E>](#). La prioridad asociada a un elemento puede cambiarse [Libro, p334]. El coste de obtener el elemento de mayor prioridad debe ser bajo.

Ejemplos:

- Cola del supermercado en la que tiene preferencia algún colectivo.
- Llegadas a urgencias en un hospital.
- Pasajeros en lista de espera.

8.1 The Priority Queue Abstract Data Type

8.1.1 Keys, Priorities, and Total Order Relations

Las colas con prioridad son TADs que almacenan **entradas** ('entries') formadas por una prioridad representada mediante una **clave** ('key') y por un elemento o **valor** ('value'). En ocasiones nos referiremos a las entradas como "pares clave-valor". Las claves y valores van por

separado. La aplicación determina la asignación de claves a valores. En algunas aplicaciones la clave puede ser un atributo del valor (p.ej., el ISBN de un libro), pero debe tener un interfaz o clase aparte ('public interface ISBN {...}') para poder ser pasada como parámetro genérico.

Dos entradas distintas pueden tener la misma clave o el mismo valor. (El mismo valor puede aparecer en entradas distintas con distinta clave y la misma clave puede aparecer en entradas distintas con distintos valores). Si no se dice lo contrario, podría haber entradas repetidas.

Notación: 'k' para claves y 'v' para valores.

Hay un **orden total** (K, \leq), también llamado **orden lineal**, sobre el conjunto de claves K (entendido como conjunto matemático, en la sección [Interfaces Comparable<T> y Comparator<T>](#) vemos cómo se puede realizar un orden total en Java).

La siguiente definición elabora la que hay en [Transparencia 3] y en [Libro, p335]: Hay una relación de igualdad en K computable: se puede determinar de forma efectiva si una clave es igual a otra.

Además:

- Totalidad: $k_1 \leq k_2$ o $k_2 \leq k_1$.
- Antisimétrica: Si $k_1 \leq k_2$ y $k_2 \leq k_1$ entonces $k_1 = k_2$.
- Transitiva: Si $k_1 \leq k_2$ y $k_2 \leq k_3$ entonces $k_1 \leq k_3$.

Totalidad y antisimétrica implican reflexiva: $k \leq k$.

Todo orden total lleva asociado un **orden total estricto** ($K, <$):

- $k_1 < k_2$ si y sólo si $k_1 \leq k_2$ y $k_1 \neq k_2$.

(Se puede definir éste orden porque tenemos igualdad).

En la práctica se usarán conjuntos K en el que hay una clave mínima y también o bien una clave máxima o bien un supremo (conjuntos discretos, finitos o enumerables).

- Clave mínima k_{\min} : $k_{\min} \leq k$, para toda clave k .
- Clave máxima k_{\max} : $k \leq k_{\max}$, para toda clave k .
- Supremo: se toma la unión de K con un conjunto formado por un único elemento $\{\text{sup}\}$ tal que $k \leq \text{sup}$ para toda k .

Convención: la clave establece la prioridad inversamente, **cuanto menor sea la clave mayor la prioridad**.

8.2.1 Entries and Comparators

Interfaz Entry<K,V>

- [Transparencia 4]
- Código: Entry.java.

Composition pattern.

Clase: producto cartesiano con “getters”.

Objeto: un par del producto cartesiano.

Interfaces `Comparable<T>` y `Comparator<T>`

- [Transparencias 5-6]

Ambos se utilizan para definir órdenes totales. La diferencia está en cómo y para qué se usan.

Interfaz `java.lang.Comparable<T>`:

`#+BEGIN_EXAMPLE`

```
public interface Comparable<T> {
    public int compareTo(T t);
}
```

`#+END_EXAMPLE`

Interfaz `java.util.Comparator<T>`:

`#+BEGIN_EXAMPLE`

Interfaz `java.util.Comparator<T>`:

```
public interface Comparator<T> {
    public int compare(T t1, T t2);
    public boolean equals(Object o);
}
```

`#+END_EXAMPLE`

Debéis leer con detalle las descripciones de la API:

- [Comparable](#)
- [Comparator](#)

Ideas claves:

- La noción de **orden natural** de `Comparable<T>`:

La implementación de `Comparable<T>` por la clase `T` hace que ésta tenga un método para comparar dos objetos de esa clase: **`o1.compareTo(o2)`**. Ese método de comparación queda fijo para `T` y se llama orden natural. El método debe devolver un entero `i` tal que:

`i < 0` si `o1` es menor que `o2`.

`i == 0` si `o1.equals(o2)`.

$i > 0$ si $o2$ es menor que $o1$.

Ejemplo: la clase Integer extiende Comparable<Integer> e implementa el método 'compareTo' usando la ALU sobre el valor 'unboxed'.

- La noción de **objeto función** de Comparator<T>:

Una implementación de Comparator<T> implementa una función de comparación 'compare' para dos objetos de la clase T: **c.compare(o1,o2)**. El método debe devolver un entero i tal que:

$i < 0$ si $o1$ es menor que $o2$.

$i == 0$ si $o1.equals(o2)$.

$i > 0$ si $o2$ es menor que $o1$.

Se pueden usar diferentes objetos comparadores para comparar objetos de T.

El método 'equals' del interfaz determina la igualdad entre **comparadores**, no entre objetos de clase T: **c1.equals(c2)**.

Ejemplo: podemos definir varios comparadores para la clase String: comparador lexicográfico, tamaño, coste de compresión bajo zip, etc.

Errores: El ejemplo en [Transparencia 6] no usa genéricos. Debería ser como se muestra a continuación:

```
#+BEGIN_EXAMPLE

public class Lexicographic implements Comparator<Point2D> {

    private int xa, ya, xb, yb;

    public int compare(Point2D a, Point2D b) {

        xa = a.getX();
        ya = a.getY();
        xb = b.getX();
        yb = b.getY();

        if (xa != xb) return (xb - xa);

        else return (yb - ya);

    }

    // no se implementa el método 'equals'

}

#+END_EXAMPLE
```

8.1.3 The Priority Queue ADT

- [Transparencia 2]
- **Código:** PriorityQueue.java, InvalidKeyException.java, EmptyPriorityQueueException.java.

Los métodos 'insert', 'min' y 'removeMin' devuelven `Entry<K,V>`. Cuando una clave no es válida ('InvalidKeyException') dependerá de cómo sea la clase K.

8.1.4 Sorting with a Priority Queue

- [Transparencias 7-13]
- También [Libro, '8.2.2 Selection-Sort and Insertion Sort', p344]

Se describen algoritmos para ordenar una lista usando una cola con prioridad.

El pseudocódigo de [Transparencia 7] usa una secuencia. Ver [6.4.2 Sequences](#). Entendemos secuencia como una lista arbitraria.

Sea E la clase de los elementos de la lista y se tienen objetos `Comparable<E>` o `Comparator<E>`. Las entradas de la cola con prioridad son de clase `Entry<E,E>` o `Entry<E,Object>`: se usan los elementos mismos como claves, la segunda componente no importa, así que podemos usar los elementos otra vez u otra clase arbitraria para valores, por ejemplo `Object`, teniendo así entradas con claves E y valores `Object` todos null. (Nótese la línea 'P.insert(e,0)' en el pseudocódigo de [Transparencia7]. El valor '0' es implementado por 'null' en Java.)

Selection-sort

Se ordena una lista S usando una cola con prioridad implementada mediante una lista no ordenada P.

- Se insertan los elementos de S en P. La inserción puede realizarse tomando elementos del final o del principio de S e insertando elementos al final o al principio de P. En ambos casos la complejidad es $O(1)$. En [Transparencia 10] se toman elementos al principio de S y se insertan al final de P, quedando P como copia de S.
- Mientras P no esté vacía, se selecciona el mínimo de P y se inserta al final de S. El método 'min' selecciona el mínimo.
- El nombre del algoritmo viene de que el trabajo se realiza en la **selección** en P del mínimo.
- Complejidad: $O(n^2)$.

Insertion-sort

Se ordena una lista S usando una cola con prioridad implementada mediante una lista ordenada P.

- Se insertan los elementos de S, tomados de cualquier forma, dentro dentro de P en orden, quedando P ordenada.

- Mientras P no vacía, se toma el mínimo de P y se inserta al final de S.
- El nombre del algoritmo viene de que el trabajo se realiza en la **inserción** en P de elementos de forma ordenada.
- Complejidad: $O(n^2)$.

Ejercicio: Implementar en Java el algoritmo de la [Transparencia 7] para *IndexList* y *PositionList*.

Ejercicio: ¿Mejora la complejidad de insertion-sort con la implementación descrita en la [Transparencia 13]?

Resumen de complejidad de colas con prioridad mediante listas

Lista ordenada:

insert	$O(n)$
min	$O(1)$
removeMin	$O(1)$

Lista desordenada:

insert	$O(1)$
min	$O(n)$
removeMin	$O(n)$

Ordenación de listas usando cola con prioridad implementada mediante lista:

- Insertion-sort: $O(n^2)$.
- Selection-sort: $O(n^2)$.

8.2 Implementing a Priority Queue with a List

8.2.1 A Java Priority Queue Implementation Using a List

- **Código:** DefaultComparator.java, SortedListPriorityQueue.java.

Lista ordenada: se inserta la entrada en la posición de la lista que le corresponde según el orden de claves.

Atributos:

- PositionList de entradas.
- Comparador.
- Position cuyo elemento es una entrada (se usa en la inserción).

Clase anidada:

- `MyEntry<K,V>` que implementa `Entry<K,V>`, con constructor y `'toString'`.

Tres constructores:

1. Construye una cola vacía. La cola usará el comparador por defecto `'DefaultComparator'` [ver código] que se basa en el orden natural de `Comparable<K>`. Nótese el upcasting y la excepción `'ClassCastException'`.
2. Construye una cola vacía pero toma un comparador.
3. Construye una cola a partir de una lista ordenada y un comparador.

Se puede cambiar el comparador de una cola: método `'setComparator'`.

La inserción la realiza el método auxiliar protegido `'insertEntry'`. El atributo `'actionPos'` referencia el nodo donde se ha insertado (`'insertion position'`). ¿Qué sentido tiene este atributo?

El método `'checkKey'` comprueba que una clave es válida. El requisito de validez aquí es que el comparador "tenga sentido" lo cual se determina solamente comprobando que una clave es igual a sí misma usando el `'compare'` del comparador.

Se implementa `'toString'`.

Ejercicio: *¿Se permite la inserción de entradas duplicadas?*

Colas con prioridad: ejemplo de uso

Clientes representados mediante cadenas de caracteres esperando turnos.

```
#+BEGIN_EXAMPLE
```

```
public static void main(String args[]) {
    PriorityQueue<Integer,String> q = new
    SortedListPriorityQueue<Integer,String>();

    ServidorClientes<String> s = new ServidorClientes<String>();

    q.insert(1, "Fulano");
    q.insert(4, "Mengano");
    q.insert(3, "Futano");
    q.insert(0, "Con Prisa");
    while (q.size() > 0) {
        s.servirCliente(q.min());
        q.removeMin();
    }
}
```

#+END_EXAMPLE

Ejercicio: *¿Se podrían mostrar los valores en la cola sin tener que invocar 'removeMin'?*

Ejercicio: *¿Tendría sentido definir un iterador para colas con prioridad? En caso positivo, implementarlo para SortedListPriorityQueue y reescribir el ejemplo de código anterior para que use el iterador.*

Los programas que usan colas con prioridad:

- Definirán interfaces o clases para claves y valores.
- Crearán objetos claves y valores.
- Podrán definir varios comparadores para claves.
- Construirán una o varias colas.
- Invocarán los métodos de la cola según la aplicación.

Esquema de uso: pasajeros en lista de espera

```
public interface Hora {

    // hora de llegada de un pasajero al mostrador de lista de espera
}

public interface DNI {

    // TAD de DNI de pasajeros
}

Comparator<Hora>

PriorityQueue<Hora,DNI> q = new SortedListPriorityQueue<Hora,DNI>()
```

Viernes 01-04-2011

8.3 Heaps

Material

- Libro: Sección 8.3.
- Transparencias: heaps-fim.pdf (sólo la mencionadas en el guión). Ignoramos las transparencias del libro: heap.pdf.
- **Código:**
 - o Interfaces: CompleteBinaryTree.java.
 - o Clases: ArrayListCompleteBinaryTree.java, HeapPriorityQueue.java.

Motivación de los montículos

Recordamos el [Resumen de complejidad de colas con prioridad mediante listas.](#)

El **montículo** ('heap') es un **TAD** que se usa para implementar colas con prioridad obteniendo las siguientes complejidades:

Insert	$O(\log n)$
Min	$O(1)$
removeMin	$O(\log n)$

Es un TAD. Puede implementarse en términos de otros TADs: árboles binarios (con ciertas propiedades especiales), IndexList, etc. También mediante implementaciones: arrays, ArrayIndexList, etc. Se suele **describir** usando árboles binarios con ciertas propiedades especiales.

La implementación del TAD montículo debe satisfacer las complejidades de la tabla anterior. Ver [Resumen de principios fundacionales](#).

Árboles Binarios (Casi)Completos

Antes de estudiar montículos estudiaremos los árboles binarios (casi)completos. En el libro el material de este apartado está repartido entre:

- '8.3.2 Complete Binary Trees and Their Representation'
- '8.3.1 The Heap Data Structure'.

No seguiremos la estructura del libro.

Repasamos definiciones y terminología de árboles generales y binarios:

- [General Trees: Terminología y Definiciones](#).
- [Binary Trees: Terminología y Definiciones](#).

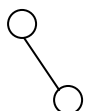
En particular: nodo interno y externo, altura, nivel, árbol binario propio y árbol binario perfecto.

[Libro, p347] Definición de **nodo a la izquierda** para árboles binarios: Un nodo 'v' está a la izquierda de otro 'w' si 'v' aparece a la izquierda de 'w' en un recorrido en inorden.

El libro da esta definición para nodos 'v' y 'w' **en el mismo nivel**. Por tanto también da la siguiente definición equivalente: existe un nodo 'u' tal que el nodo 'v' está en el subárbol izquierdo de 'u' y el nodo 'w' está en el subárbol derecho de 'u'.

La definición mediante posición en el recorrido en inorden es más general y podría aplicarse a nodos en cualquier nivel. Este no es el caso para la definición en función de un ancestro común. Contraejemplo:

#+BEGIN_EXAMPLE



#+END_EXAMPLE

El nodo raíz aparece a la izquierda del hijo derecho en un recorrido en inorden pero no hay un nodo 'u' tal que la raíz está en su subárbol izquierdo.

Definición de árbol binario completo ('complete binary tree')

Algunos autores lo definen como sinónimo de árbol binario perfecto [CLRS'01] [ALT'03] definen **árbol binario casi-completo** para lo que el [Libro, p347] y [Transparencia 4] definen como árbol binario completo, lo cual puede tener más sentido. Como compromiso entre ambas definiciones usaremos **árbol binario (casi)completo** en este guión.

Informalmente, se trata de un árbol de altura h que es perfecto hasta el nivel $h-1$ y que en el nivel h se va completando de izquierda a derecha.

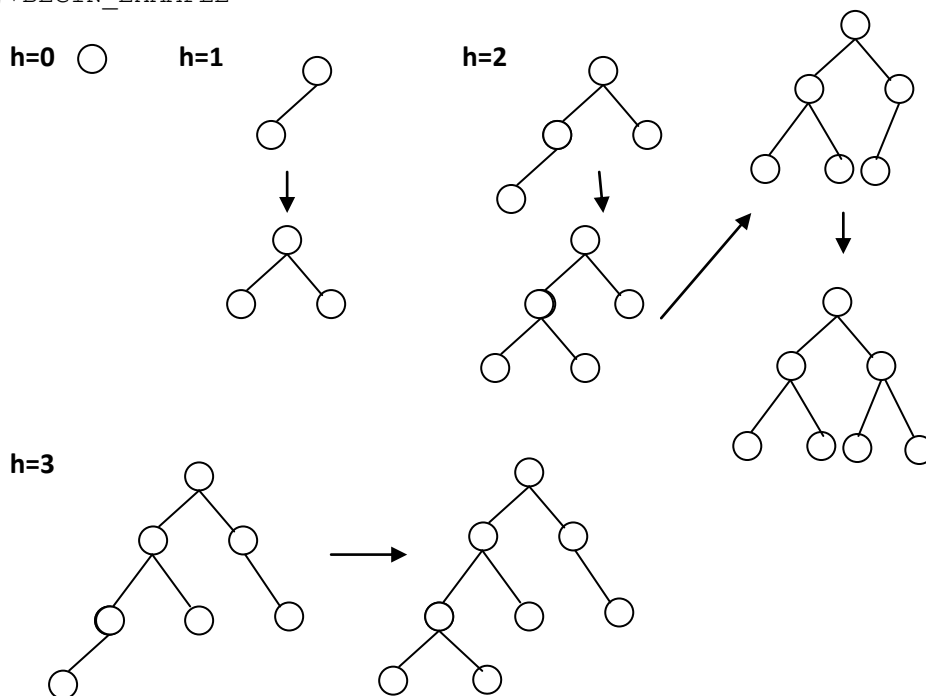
[Libro, p347] y [Transparencia 4]:

1. Los niveles 0 a $h-1$ tienen el máximo número de nodos: 2^n nodos en el nivel n . (Es perfecto hasta nivel $h-1$.)
2. En el nivel $h-1$ los nodos internos están a la izquierda de los nodos externos. Esto debe interpretarse así: en el nivel $h-1$, si hay hojas en ese nivel entonces los nodos internos de ese nivel (debe haberlos) están a la izquierda de las hojas
3. Hay un único nodo con un solo hijo que debe ser un hijo izquierdo. **Errores:** Esta propiedad no está en [Transparencia 4].

El árbol (casi)completo se va llenando de izquierda a derecha en el último nivel hasta convertirse en perfecto, continuándose el llenado en un siguiente nivel.

Ejemplo:

#+BEGIN_EXAMPLE



#+END_EXAMPLE

Definición de **último nodo de un árbol (casi)completo**: la hoja más a la derecha que está en el nivel h . [Libro, p348]: El nodo en el nivel h tal que todos los otros nodos de nivel h están a la izquierda. Se trata del último nodo insertado.

Errores: El [Libro, p347] lo define como "the rightmost, deepest external node" lo cual es ambiguo y puede leerse como "la hoja más a la derecha de mayor profundidad" (incorrecto como en [Transparencia 4], "rightmost node of maximum depth"). Debe leerse como "la hoja de mayor profundidad que está más a la derecha" u "hoja más profunda más a la derecha".

La **altura de un árbol (casi)completo** es $O(\log n)$. Demostración diferente a la de [Transparencia 5]:

- Hasta el nivel $h-1$ es un árbol perfecto con $2^{((h-1)+1)-1} = 2^{h-1}$ nodos.
- En el nivel h hay como mínimo 1 nodo y como máximo 2^h nodos.
- Total mínimo: $2^h - 1 + 1$ nodos.

$$\begin{aligned}
 2^h &\leq n \\
 \Leftrightarrow \{ \text{aplicando log en ambos lados} \} \\
 \log(2^h) &\leq \log(n) \\
 \Leftrightarrow \{ \text{Definición de log} \} \\
 h &\leq \log(n)
 \end{aligned}$$

Por tanto, h es $O(\log n)$.

Ejercicio: Indicar la relación entre un árbol binario (casi)completo y un [árbol binario equilibrado](#).

TAD de Árboles (Casi)Completo

Interfaz CompleteBinaryTree<E>

Código: CompleteBinaryTree.java.

Extiende el [Interfaz BinaryTree<E>](#) con los siguientes métodos:

- 'public Position<E> add(E elem)': Inserta un nodo hoja con el elemento 'elem' en el árbol binario (casi)completo de forma que dicho nodo es el "último nodo" tal y como se ha definido en [Árboles Binarios \(Casi\)Completo](#). El método devuelve el nuevo nodo insertado.
- 'public E remove()': Borra el último nodo y devuelve su elemento.

Representación de Árboles (Casi)Completo con Arrays

La definición de árbol (casi)completo sugiere una representación del árbol mediante arrays. El árbol es perfecto en los niveles 0 a $h-1$ y en el nivel h se llena de izquierda a derecha con lo que se pueden meter todos los nodos **consecutivamente** en un array 'v' de tamaño $n+1$:

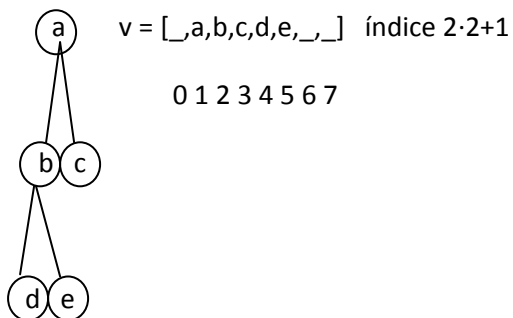
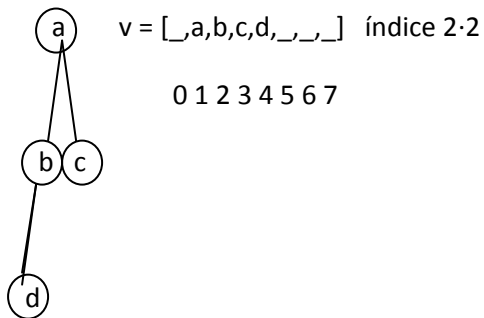
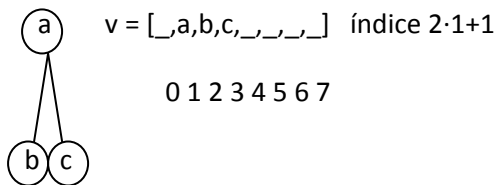
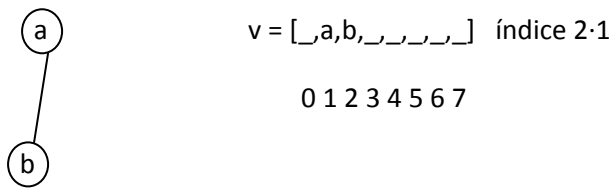
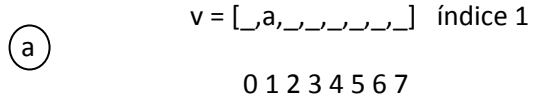
- La raíz está en $v[1]$.
- El nodo en $v[i]$ tiene su hijo izquierdo en $v[2 \cdot i]$, su hijo derecho en $v[2 \cdot i + 1]$ y su padre (si $i \neq 1$) en $v[i/2]$ (división entera).
- $v[0]$ se deja vacío. ¿Por qué?

En [CLRS'01](#) se definen las siguientes funciones sobre índices:

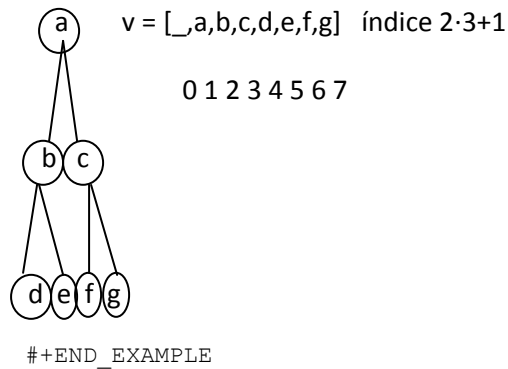
- $\text{parent}(i) = i/2$
- $\text{left}(i) = 2 \cdot i$
- $\text{right}(i) = 2 \cdot i + 1$

Ejemplo:

#+BEGIN_EXAMPLE



...



Los métodos 'add' y 'remove' acceden al último nodo del árbol:

- El último nodo está en el índice n donde n es el número de nodos.
- Por tanto 'add' añadirá un nodo en el índice $n+1$ y 'remove' borrará el nodo en el índice n .
- Ambos métodos tienen complejidad $O(1)$.

Clase *ArrayListCompleteBinaryTree<E>*

Código: ArrayListCompleteBinaryTree.java.

Errores: Los comentarios al comienzo usan [rank](#) pero se refieren a índices.

Implementa *CompleteBinaryTree<E>* mediante un *ArrayList* de la JCF (no mediante *IndexList* [6.1 Array Lists](#)). Los elementos almacenados en el *ArrayList* son objetos de clase *BTPos<E>*. El *ArrayList* es el único atributo de la clase *ArrayListCompleteBinaryTree<E>*.

La clase *BTPos<E>* implementa *Position<E>* (nodo del árbol). Tiene dos atributos, el elemento y el índice del elemento en el *ArrayList*.

Recordamos que los métodos de [Interfaz Tree<E>](#) e [Interfaz BinaryTree<E>](#) heredados por *CompleteBinaryTree<E>* trabajan con *Position<E>* y no con índices. Pero los índices se necesitan para implementar todo lo descrito en [Representación de Árboles \(Casi\)Completos con Arrays](#), de ahí que *BTPos<E>* guarde el índice de un nodo.

La clase *BTPos<E>* que implementa *Position<E>* es en esta ocasión una clase anidada y no una clase separada como en los árboles generales y binarios que hemos visto en:

- [Interfaces y clases de la implementación de Tree<E>](#).
- [Interfaces y clases de la implementación de BinaryTree<E>](#).

BTPos<E> ofrece un constructor, dos “getters”, un “setter” y 'toString'. Los métodos 'isExternal', 'isRoot', 'hasLeft', 'hasRight', 'root', 'left', 'right' y 'parent' se definen usando la lógica de la representación en un array. Ver [Representación de Árboles \(Casi\)Completos con Arrays](#).

Se define el método 'checkPosition' que comprueba que un objeto de clase Position<E> no es null y es de clase BTPos (no BTPos<E>, de nuevo lo de 'type erasure').

Se define el método 'iterator' (aunque ArrayListCompleteBinaryTree<E> no extiende Iterable<E>). Ver [Cómo iterar sobre TADs](#).

Se define el método 'positions' que es lo que hemos llamado 'elements' en [Cómo iterar sobre TADs](#).

Se implementan 'add' y 'remove' de [Interfaz CompleteBinaryTree<E>](#).

Se define un nuevo método observador 'sibling'.

Se define 'replace' que se usa para la implementación de montículos [Montículos](#).

Se definen otros métodos como 'swapElements' que no se usan en ninguna otra parte del paquete.

Errores: Estos métodos deberían haberse definido en sus clases o interfaces correspondientes.

Montículos

Definición de **montículo ('heap')**: árbol binario (casi)completo que almacena Entry<K,V> (Ver [Interfaz Entry<K,V>](#)) en los nodos tal que para todo nodo distinto de la raíz, la clave de su entrada es mayor o igual que la clave de la entrada almacenada en el nodo padre. La clave **menor** está en la raíz. Las claves en el camino de la raíz a las hojas están en orden creciente.

Ilustración en [Transparencia 6].

Usando la lógica de índices y las funciones sobre índices definidas en [Representación de Árboles \(Casi\)Completos con Arrays](#):

```
v[i].getKey() >= v[parent(i)].getKey()
```

[Libro, p346] llama a esta propiedad 'heap-order property'. El nombre "montículo" viene de amontonar claves por orden creciente.

Se puede definir un montículo para que la clave **mayor** esté en la raíz. Basta con definir un comparador que invierta el orden. Estos montículos se llaman 'max-heaps'. La definición de arriba es para 'min-heaps' [CLRS'01,p129].

8.3.3 Implementing a Priority Queue with a Heap

Se implementan los métodos del interfaz PriorityQueue<E> definido en PriorityQueue.java. Ver [8.1.3 The Priority Queue ADT](#).

El método 'min' devolverá la entrada en el nodo raíz en O(1).

El método 'insert' [Transparencias 7-8]:

1. Inserta el nuevo nodo como último nodo del árbol binario (casi)completo en $O(1)$ (ver [Representación de Árboles \(Casi\)Completos con Arrays](#)). El árbol resultado sigue siendo (casi)completo.
2. Se comprueba que la 'heap-order property' se cumple entre el nuevo nodo y su padre.
3. Si no se cumple se intercambian las entradas entre el nodo y su padre. Se repite la comprobación en el siguiente nivel hacia arriba, con el padre y el abuelo, hasta que se cumpla la 'heap-order property' o se llegue a la raíz.

[Libro, p356] A este proceso se le llama 'up-heap bubbling', abreviado 'upheap'. Tiene complejidad $O(\log n)$ pues la altura de un árbol (casi)completo es $O(\log n)$.

Por tanto, la complejidad de 'insert' es $O(\log n)$

El método 'remove' [Transparencias 9-10]:

1. Debe eliminar la entrada de menor clave, pero dicha entrada está en la raíz que no puede eliminarse. En su lugar, se mueve la entrada del último nodo del árbol binario (casi)completo a la raíz en $O(1)$ y después se elimina el último nodo en $O(1)$. Ver [Representación de Árboles \(Casi\)Completos con Arrays](#).
2. Se comprueba la 'heap-order property' entre la raíz y su hijo izquierdo (si la raíz no tiene hijo derecho) o su hijo de menor clave (si la raíz tiene hijo derecho). Se repite la comprobación en el siguiente nivel hacia abajo hasta que se cumpla la 'heap-order property' o se llegue a una hoja.

[Libro, p356] A este proceso se le llama 'down-heap bubbling', abreviado 'downheap'. Tiene complejidad $O(\log n)$ pues la altura de un árbol (casi)completo es $O(\log n)$.

Por tanto, la complejidad de 'remove' es $O(\log n)$.

Ejercicio: ¿Por qué en 'downheap' se elige el hijo de menor clave?

Errores: Si la implementación del montículo se realiza mediante estructuras enlazadas y no mediante arrays, encontrar el último nodo puede seguir teniendo complejidad $O(1)$ y no $O(\log n)$ como se dice en [Transparencia 11]. Pueden tenerse atributos de la clase que implementa el montículo que sean referencias al último nodo y a su padre.

8.3.4 A Java Heap Implementation

Código: HeapPriorityQueue.java.

La clase HeapPriorityQueue<E> implementa PriorityQueue<E>.

Dos atributos, el árbol (casi)completo de entradas y el comparador de claves.

Se implementa el [Interfaz Entry<K,V>](#) mediante una clase anidada MyEntry<K,V>.

Dos constructores. Ambos crean un árbol (casi)completo vacío implementado con la [Clase ArrayListCompleteBinaryTree<E>](#). Un constructor usa el comparador por defecto y el otro el comparador pasado como argumento.

Los métodos interesantes son 'upheap', 'downheap' y 'swap'.

Ejercicio: *¿Podría definirse el método 'swap' usando el método 'swapElements' de ArrayListCompleteBinaryTree<E>? Observad que la lógica de índices se ha realizado de forma abstracta, utilizando los métodos de ArrayListCompleteBinaryTree<E>.*

8.3.5 Heap Sort

[Transparencias 12-14]

Viernes 08-04-2011

9.1 Maps

Material

- Libro: Sección 9.1.
- Transparencias: maps.pdf.
- Código:
 - Interfaces
 - Map.java.
 - Clases
 - InvalidKeyException.java.

Repasar: [8.2.1 Entries and Comparators](#).

Motivación de las funciones finitas

Map significa 'correspondencia' o 'función'. Las funciones finitas se usan para gestionar información (valores) a través de claves. La información puede ser compleja y no tener propiedades que permitan gestionarla rápidamente. Se usan claves que sí tendrán esas propiedades.

La inserción, búsqueda y borrado se hará a través de las claves. Ejemplo: Base de datos de alumnos. Se usa el NIF/NIE como clave.

La asociación de claves a valores [Interfaz Entry<K,V>](#) se ha introducido en el tema de colas con prioridad (Ver [8 Priority Queues](#) y se ha usado en el tema de montículos (Ver [8.3 Heaps](#)) que implementan colas con prioridad. Sin embargo, en las colas con prioridad:

- El objetivo es obtener el elemento de mayor prioridad rápidamente.
- Las claves se usan como prioridades.

Un "map" es el grafo (extensión) de una **función finita**: colección finita de entradas (pares clave-valor) que poseen una **propiedad funcional** que puede expresarse informalmente en palabras de diversas maneras equivalentes:

- Todas las entradas tienen claves distintas.
- No hay entradas con la misma clave.
- La clave asociada a un valor dado es única.
- Si algunas entradas tienen claves iguales entonces los valores asociados también son iguales.

Definición matemática de función finita

Definición en teoría de conjuntos:

- Dados un conjunto de claves K y un conjunto de valores V .
- Definimos el producto cartesiano de K y V por comprensión:
 - o $K \times V = \{ (k,v) \mid k \text{ está en } K \text{ y } v \text{ está en } V \}$.
- Una función M con origen ('source') K y destino ('target') V es un subconjunto de $K \times V$ que cumple la siguiente propiedad funcional:
 - o Para toda k_i y k_j en el dominio de M , $k_i = k_j \Rightarrow M(k_i) = M(k_j)$.
- Requisito mínimo sobre K : debe proporcionar una operación de igualdad entre todos sus elementos.
- Dominio de M :
 - o $\{ k \mid k \text{ está en } K \text{ y } (k,v) \text{ está en } M \text{ para algún } v \}$.
- Recorrido o Imagen de M :
 - o $\{ v \mid v \text{ está en } V \text{ y } (k,v) \text{ está en } M \text{ para algún } k \}$.
- Decir " (k,v) está en M " es lo mismo que decir " $M(k)=v$ ".
- Una función es finita cuando su dominio es finito.

Implementación inmediata: lista de entradas sin repeticiones. También llamadas 'association lists' (Ver [A Simple List-Based Map Implementation](#)).

Las implementaciones más eficientes explotarán propiedades de las claves: orden, dispersión ('hash'), indización, etc. [Libro, p382]: "the key associated with an object determines its "location" in the data structure".

ERRORES: [Libro, p382] "In order to achieve the highest level of generality, we allow both the keys and the values stored in a map to be of any object type". Esto contradice el [Requisito mínimo sobre K](#).

¿Cómo se realiza el [Requisito mínimo sobre K](#) en Java? [JP, p104]:

1. Si K es una clase envoltura ('wrapper class') de un tipo primitivo (p.ej., clases Integer, Character, Float, etc.) entonces gracias al "autoboxing" se pueden usar los operadores de comparación '==', '!=', etc, siempre que estén definidos para dicho tipo primitivo.
2. Si K es una clase, hereda de Object y puede implementar el método 'equals'. Si K es un interfaz, la clase que la implementa hereda de Object y puede implementar 'equals'.

3. Si K es una clase, puede implementar el interfaz Comparable<K>. Si K es un interfaz, puede heredar (extender) el interfaz Comparable<K>. La clase que implemente el interfaz K deberá implementar los metodos de Comparable<K>.
4. Se puede implementar un objeto Comparator<K>.

CUIDADO: En este gui3n as3 como en el libro y las transparencias las ecuaciones con claves ($k_1 = k_2$, $k_1 \neq k_2$) deben entenderse en un sentido matemático. En Java la igualdad debe realizarse seg3n alguna de las cuatro formas antes enumeradas.

9.1.1 The Map ADT

- **C3digo:**
 - o Map.java
 - o InvalidKeyException.java.
- [Transparencia 3]
- Se define el Interfaz Map<K,V>.

M3todos de inserci3n ('put'), b3squeda ('get') y borrado ('remove').

- Buscar el valor asociado a una clave es aplicar la funci3n finita:
 - o En matemáticas: $M(k)$.
 - o En Java: `M.get(k)`.
- ERRORES: [Transparencia 3]. La descripci3n de 'put' no es correcta. Es correcta en [Libro, p383] y en Map.java.
- Lanzas 'InvalidKeyException' y adem3s usan 'null' para notificar que se ha producido una situaci3n excepcional. El uso de 'null' tiene varias consecuencias:
- El valor 'null' tiene diversos significados que dependen del contexto. Por ejemplo, problemas con entradas (`k,null`).
- Debe preguntarse antes ('isEmpty', '`!= null`', etc):

Mal:

```

#+BEGIN_EXAMPLE
        c = M.get(2).toLowerCase();           // Posible excepci3n.
        c = M.put(2, 'A').toUpperCase();     // Posible excepci3n.
#+END_EXAMPLE

```

Bien:

```

#+BEGIN_EXAMPLE
        if (!M.isEmpty() && (c = M.get(2)) != null)
            c = c.toLowerCase();

        if (c = M.put(2, 'A')) != null)
            c = c.toUpperCase();
#+END_EXAMPLE

```


Código estructurado. Métodos no componibles. Más rápido.

- No usar bloques try-catch-finally:

```
#+BEGIN_EXAMPLE

    try {

        c = M.get(2).toLowerCase();

    } catch (NullPointerException e) {

        /* Aquí tendría que ir la rama 'else' */

    }

#+END_EXAMPLE
```

Código desestructurado. Métodos no componibles. Más lento.

- El constructor de 'InvalidKeyException' toma una cadena de caracteres que indicará por qué la clave no es válida.
- Hay otra solución a la dicotomía entre excepción o centinela ('null') consiste en extender el "destino" de los métodos. No exploraremos esta solución por falta de tiempo.

El Map de JCF usa de igual forma excepciones y 'null'. Métodos 'keySet', 'values' y 'entrySet' que devuelven respectivamente iterables de claves, valores y entradas. El método 'entrySet' es lo que hemos llamado 'elements' en [Cómo iterar sobre TADs](#).

ERRORES: [Libro, p383] y [Transparencia 3] dicen 'iterable collection'. Debe entenderse 'collection' informalmente como iterable, no como Collection<E>.

Ejercicio: Implementar 'entrySet' en términos de 'keySet' y 'get'.

Ejercicio: Implementar 'keySet' y 'values' en términos de 'entrySet'.

9.1.2 A Simple List-Based Map Implementation.

- [Transparencia 4]

Una función finita vacía no es representada por el valor 'null'. La función finita vacía será construida por el constructor de la clase que implemente Map<K,V>, el cual devolverá un objeto en memoria. En [Libro, p384] y [Transparencia 4] se usa el símbolo de conjunto vacío para describir una función finita vacía.

Se sugiere usar una PositionList (desordenada) sin entradas repetidas.

En [Transparencia 4] la entradas se insertan al final de la lista. En una lista desordenada no importa dónde se inserta. Podría insertarse al principio o en cualquier otra posición.

Ejercicio: ¿Tendría sentido usar IndexList?

ERRORES: [Libro, p385] y [Transparencia 5] insisten en una lista doblemente enlazada ('doubly-linked list'). Sin embargo, esto es una decisión de implementación. La [Clase NodePositionList<E>](#) se implementa mediante una lista doblemente enlazada, pero se podría usar una clase que implementase el [Interfaz PositionList<E>](#) mediante una lista simplemente enlazada. Las operaciones del interfaz Map<K,V> no fuerzan una u otra implementación.

[Transparencias 6-8] Pseudocódigo de los métodos 'get', 'put', y 'remove' para una función finita implementada mediante una PositionList desordenada.

- ERRORES: En los tres casos se itera sobre S.positions() donde S es la lista. El comentario de la asignación 'B = S.positions()' en [Transparencia 6] dice "B is an iterator of the positions in S". Sin embargo, S.positions() no devuelve un Iterator<Position<E>> sino un Iterable<Position<E>> (ver código de [Interfaz PositionList<E>](#)). Por tanto, la asignación debería ser 'B = S.positions().iterator()'.
 - No se itera sobre la lista sino sobre el iterable devuelto por S.positions(), que puede ser cualquier otra estructura de datos iterable. En la implementación [Clase NodePositionList<E>](#) el método 'positions' devuelve otra NodePositionList. Por tanto si (en tiempo de ejecución) S es una NodePositionList entonces S.positions() es una NodePositionList. Pero S podría ser (en tiempo de ejecución) otra implementación de PositionList. (Nótese que la "lista" se llama 'S', lo que sugiere que realmente se trata de una secuencia (Ver [6.4.2 Sequences](#))). Dicho esto, los iteradores abstraen la estructura del objeto sobre el que iteran. Por tanto, el único factor relevante es que iterar sobre lo que devuelva S.positions() tiene la misma complejidad que iterar sobre S: O(n).
 - Pseudocódigo de get(k) usando el iterador de la lista y sin salir del bucle mediante return (Ver [Corolario para bucles](#)).

```

#+BEGIN_EXAMPLE

Algorithm get(k)
    B = S.iterator()
    found = false
    while (B.hasNext() and not found) do
        p = B.next()
        found = p.element().getKey() == k
    { En este punto se cumple 'not B.hasNext() or found' }
    if found then return p.element().getValue()
    else return null

#+END_EXAMPLE

```

- En los tres casos hay que recorrer la lista y para ello deben usarse iteradores. Si usamos otros métodos del [Interfaz PositionList<E>](#) nos quedará un código más complicado. Por ejemplo:

```

#+BEGIN_EXAMPLE

```

```

Algorithm get(k)
  if S.isEmpty() then return null
  else
    found  = false
    end    = false
    cursor = S.first()
  do
    found = cursor.element().getKey() == k
    end   = cursor == S.last()
    if not end then
      cursor = S.next(cursor)
  while(not found and not end)
  { En este punto se cumple 'found or end' }
  if found then return cursor.element().getValue()
  else return null
#+END_EXAMPLE

```

Con un iterador se abstrae todo lo relativo al uso del cursor.

- **Ejercicio:** Escribir el pseudocódigo (o implementar en Java) el método 'remove' de la función finita usando el iterador de la lista. Recuérdese que debe utilizarse [el método 'remove' del iterador](#).
- **Ejercicio:** Implementar en Java los métodos 'get', 'put' y 'remove' utilizando una lista *S* que en vez de *PositionList<E>* es *List<E>* (interfaz definido en [Listas: comentarios y ejercicios](#)). Implementar dichos métodos sin usar los iteradores o iterables de *List<E>*. Comparar el resultado con el pseudocódigo mostrado en los puntos anteriores.

Complejidades: [Transparencia 9].

Maps en la JCF

[Libro, p384]

Interfaz `java.util.Map<K,V>` que usa interfaz `java.util.Map.Entry<K,V>`.

9.2 Hash tables

Material

- Libro: Sección 9.2.
- Transparencias: hashtables.pdf.
- Código:
 - Interfaces:
 - Map.java.
 - Clases:
 - InvalidKeyException.java
 - HashMap.java.

Motivación de tablas de dispersión

Implementar funciones finitas de forma que la complejidad de la inserción, búsqueda y borrado sea (en muchas ocasiones) $O(1)$.

Las claves deberán tener la propiedad de ser "dispersables". Los valores pueden ser cualquier cosa (Ver [Motivación de las funciones finitas](#)).

Definición de tablas de dispersión

[Transparencia 3]

1. Constante de dos elementos:
 1. Un array T (que llamamos tabla) de tamaño N que almacena entradas.
 2. Una función (método) de dispersión ('hash function')

$$h : K \rightarrow [0..N-1]$$

Cuyo objetivo es dispersar las claves dentro del intervalo $[0..N-1]$.

La función de dispersión se suele definir mediante la composición de dos funciones $h(k) = h_2(h_1(k))$:

1. $h_1 : K \rightarrow \text{Integer}$, función de codificación ('hash code').
2. $h_2 : \text{Integer} \rightarrow [0..N-1]$, función de compresión y dispersión ('compression function').

El objetivo es almacenar la entrada (k,v) en $T[h(k)]$.

Importante (y no se dice en el libro): computar $h(k)$ debe ser $O(1)$

En la práctica el número de claves posibles es mucho mayor que N con lo que la función h no será inyectiva.

- No inyectiva: no se cumple $h(k_1) = h(k_2) \Rightarrow k_1 = k_2$.

No $(h(k_1) = h(k_2) \Rightarrow k_1 = k_2)$

$\Leftrightarrow \{ \text{Definición de implicación} \}$

No $(\text{no}(h(k_1) = h(k_2)) \text{ o } k_1 = k_2)$

\Leftrightarrow { De Morgan para la disyunción }

$\text{No}(\text{no}(\text{h}(k_1) = \text{h}(k_2))) \text{ y } \text{no}(k_1 = k_2)$

\Leftrightarrow { Doble negación }

$\text{h}(k_1) = \text{h}(k_2) \text{ y } \text{no}(k_1 = k_2)$

- En palabras, a dos claves diferentes h les asigna el mismo índice en la tabla. A esto se le llama **colisión**.

El objetivo del estudio matemático de las funciones de dispersión es encontrar funciones que dispersan bien y tienen pocas colisiones.

Ejercicio: *¿Tendría sentido que la tabla fuera una `IndexList` en vez de un `array`?*

ERRORES: Tanto el libro como las transparencias utilizan 'entry' informalmente para referirse a veces a claves y otras veces a pares clave-valor. Los ejemplos en [Transparencia 4, 9, 11 y 15] [Libro, p394, p395] por simplificar no muestran las entradas almacenadas en la tabla sino sólo los valores. Pero debe entenderse que las tablas almacenan entradas (pares clave-valor). En Java, objetos de clases que implementan el [Interfaz `Entry<K,V>`](#).

9.2.3 Hash Codes

- Libro: Sección 9.2.3.
- [Transparencias 6-7]

La función $h_1 : K \rightarrow \text{Integer}$. Su signatura impone que K debe ser numerable.

Comentarios a las transparencias:

- 'Memory address' tiene el problema de 'aliasing': objetos diferentes (direcciones de memorias diferentes) pueden representar el mismo dato. Ejemplos: objetos de clases numéricas "boxed" como `Integer`, `Character`, `Float`, etc, así como objetos de clase `String`.

En Java, la clase `Object` define el método 'hashCode' que debe devolver un entero de 32 bits. El comportamiento por defecto es devolver la dirección de memoria del objeto. Sin embargo, la clase `String` sobrescribe ('overrides') acertadamente 'hashCode' para que no haga eso.

EJERCICIO: *Averiguar cómo define 'hashCode' la clase `Integer`.*

- 'Component sum' tiene el problema de la conmutatividad ' $X+Y = Y+X$ ' y asociatividad ' $X+(Y+Z) = (X+Y)+Z$ ' de la suma. De éstas se deduce que $X+Y+Z = Z+Y+X$. Puede producirse 'aliasing'. Ejemplo con cadenas de caracteres "amor" y "roma". Usaremos como bloques los bits del código ASCII de cada caracter. (Java codifica caracteres usando UTF-16.)

Letra	ASCII		Letra	ASCII
A	97		R	114
M	109		O	111
O	111		M	109
R	114		A	97
Suma	431		Suma	431

- 'Polynomial accumulaion' o 'polynomial hash codes': se dan pesos a cada partición antes de la suma. [Libro, p389] da una explicación menos formal: "Intuitively, a polynomial hash code uses multiplication by the constant 'a' as a way of "making room" for each component".

[Libro, p390] y [Transparencia 7] recomiendan los valores 33, 37, 39 y 41 para 'a' basándose en supuestos estudios empíricos.

Letra	ASCII	Peso		Letra	ASCII	Peso
A	97	33^3		R	114	33^3
M	109	33^2		O	111	33^2
O	111	33^1		M	109	33^1
R	114	33^0		A	97	33^0
Resultado	431	3608367		Resultado	431	4221385

Polinomios:

- "amor": $97 \cdot 33^3 + 109 \cdot 33^2 + 111 \cdot 33^1 + 114$
- "roma": $114 \cdot 33^3 + 111 \cdot 33^2 + 109 \cdot 33^1 + 97$

'Cyclic Shift Hash Codes' [Libro, p390] No viene en las transparencias. En vez de multiplicar se desplazan bits.

9.2.4 Compression Functions

- Libro: Sección 9.2.4.
- [Transparencia 8]

La función $h_2 : \text{Integer} \rightarrow [0..N-1]$. Además de compresión se desea evitar colisiones: dispersión. Método de división ('división'): $h_2(x) = x \bmod N$, con N primo.

Usa aritmética modular que dispersa circularmente:

X	0	...	N-1	N	N+1	N+2	...	N+N-1	...
$H_2(x)$	0	...	N-1	0	1	2	...	N-1	...

Método MAD: $h_2(x) = ((a \cdot x + b) \bmod p) \bmod N$,

- N primo.
- $p > N$, también primo.
- a y b están en $[0..p-1]$, con $a > 0$.
- La probabilidad de colisión es $1/N$.

ERRORES: [Transparencia 8], ' $h_2(y)$ ' no se define como hemos indicado arriba. Se define bien en [Libro, p392].

9.2.5 Collision-Handling Schemes

Separate Chaining

En Castellano: "encadenamiento". A la tabla se la suele llamar **vector de cubos** ('bucket array'). Cada posición $T[h(k)]$ de la tabla almacena una **función finita** (de nuevo, interfaz $\text{Map}<K,V>$) que suele implementarse mediante una lista (llamada "cubo") [9.1.2 A Simple List-Based Map Implementation](#). La lista almacena las claves que colisionan.

El encadenamiento es la implementación de tablas de dispersión más usada. Tiene buena eficiencia si la función de dispersión tiene pocas colisiones. [Transparencia 10] Los métodos de inserción, búsqueda y borrado delegan en los respectivos métodos de la función finita almacenada en cada posición $T[h(k)]$ de la tabla.

```
#+BEGIN_EXAMPLE
```

```
    Algorithm get(k) :
        return T[h(k)].get(k)
```

```
#+END_EXAMPLE
```

[Libro, p394] Definición de **factor de carga** ('load factor') de la tabla:

- Factor de carga: $\lambda = n/N$.
- donde n es el número total de entradas en la tabla.
- donde N es el tamaño de la tabla (del array).

Con una buena función de dispersión (dispersión uniforme), la media del tamaño de cada lista (cubo) será λ : el número de entradas que colisionan se divide por igual, n/N , entre las posiciones de la tabla. La complejidad **esperada** de la inserción, borrado y búsqueda es $O(\lambda)$:

- Computar $h(k)$ es $O(1)$.
- Acceder a la función finita en $T[h(k)]$ es $O(1)$.

- Encontrar la clave en dicha función finita (implementada como una lista) es $O(\lambda)$, con λ el tamaño de la lista.

Por tanto:

- Caso **esperado**: Una buena función de dispersión mantiene el valor de λ acotado, $\lambda < 1$. Entonces $O(\lambda) = O(1)$.
- Caso **peor**: Una mala función de dispersión no mantiene el valor de λ acotado. La complejidad es $O(n)$ ($1/N$ es un factor constante que multiplica a n y el valor de λ crece con n). Otra forma de entender el caso peor: imaginad una función de dispersión que hace que todas las claves colisionen y por tanto todas las entradas se insertan en la misma lista. La búsqueda en la tabla tiene entonces la misma complejidad que la búsqueda en una lista.

Open Addressing

Motivación del direccionamiento abierto

"Open addressing" en castellano: "direccionamiento abierto". Las claves que colisionan no se almacenan en memoria adicional (lista) sino dentro de la misma tabla, en posiciones vacías disponibles. Hay distintos métodos para seleccionar una posición vacía. Ligeramente peor que encadenamiento. Implementación adecuada cuando no se dispone de suficiente memoria.

Aumentan las colisiones: se producen colisiones entre claves que no colisionan vía h , $h(k_1) \neq h(k_2)$, pero colisionan en la tabla. Ejemplo: sean k_1 , k_2 y k_3 claves tal que:

- k_1 y k_2 colisionan vía h .
- k_3 no colisiona con k_1 (ni con k_2) vía h .
- $T[h(k_1)]$ guarda la entrada asociada a k_1 .

Al insertar k_2 puede suceder que ésta ocupe la posición en la tabla que le correspondería a k_3 : nueva colisión. Las nuevas colisiones se propagan: a su vez k_3 puede ocupar una posición que le correspondería a otras claves con las que no colisiona vía h , etc.

Métodos de sondeo ('probing')

Sondeo lineal ('linear probing')

Borrado remove(k): sea $i = h(k)$,

1. Si $T[i]$ está vacía entonces la clave no está en la tabla.
2. Si $T[i]$ está ocupada por una entrada:
 - 2.1. Si $T[i].getKey()$ y k son iguales entonces se borra la entrada y se marca esa posición de la tabla como "disponible" ('available').
 - 2.2. Si $T[i].getKey()$ y k son distintas entonces **se sondea la tabla hacia la derecha circularmente**, terminándose cuando:

- 2.2.1. Se encuentra una posición vacía, procediéndose entonces como en el punto 1.
 - 2.2.2. Se encuentra una entrada con k , procediéndose entonces como en el punto 2.1.
 - 2.2.3. Se ha recorrido toda la tabla circularmente (tabla llena), procediéndose entonces como en el punto 1.
3. Si $T[i]$ está marcada como disponible entonces se sondea la tabla como en el punto 2.2.

El movimiento circular a la derecha se expresa en términos de manipulación de índices: siendo $i=h(k)$,

$$\text{next}(i,j) = (i+j) \bmod N.$$

El movimiento circular se consigue dando valores a 'j':

$$\text{next}(i,0), \dots, \text{next}(i,N-1).$$

¿Por qué marcar posiciones borradas como disponibles y no como vacías?

Dadas k_1 y k_2 que colisionan vía h :

- Supongamos que tras insertar k_1 y k_2 tenemos la tabla:

...	K_1	K_2	...
-----	-------	-------	-----

- Se invoca $\text{remove}(k_1)$ marcándose la posición $h(k_1)$ como vacía:

...	null	K_2	...
-----	------	-------	-----

- La invocación $\text{remove}(k_2)$ falla: la búsqueda termina al encontrar la posición $h(k_2)$ vacía.

Inserción $\text{put}(k,v)$: sea $i=h(k)$,

1. Si $T[i]$ está vacía o marcada como disponible entonces se inserta la entrada en esa posición.
2. Si $T[i]$ está ocupada por una entrada:
 - 2.1. Si $T[i].getKey()$ y k son iguales entonces se actualiza la entrada.

2.2. Si $T[i].getKey()$ y k son distintas entonces **se sondea la tabla hacia la derecha circularmente**, terminándose cuando:

- 2.2.1. Se encuentra una posición vacía o marcada como disponible, procediéndose entonces como en el punto 1.
- 2.2.2. Se encuentra una entrada con la clave k , procediéndose entonces como en el punto 2.1.
- 2.2.3. Se ha recorrido toda la tabla circularmente (tabla llena).

Desventaja del sondeo lineal: las claves que colisionan acaban teniendo índices cercanos en la tabla. Se forman grupúsculos contiguos ('primary clustering') cuyo recorrido aumenta el tiempo de sondeo.

Búsqueda $get(k)$: similar al borrado sólo que en el punto 2.1 en vez de borrar la entrada se devuelve $T[i].getValue()$.

Sondeo cuadrático ('quadratic probing')

Primero generalizamos la expresión del movimiento circular a la derecha en términos de manipulación de índices: siendo $i = h(k)$,

$$\text{next}(i,j) = (i + f(j)) \bmod N$$

donde 'f' es una función fija.

En sondeo lineal: $f(j) = j$.

En sondeo cuadrático: $f(j) = j^2$.

Se evita el 'primary clustering' pero se tiene 'secondary clustering'. Informalmente, se dan saltos en el sondeo pero se terminan formando grupos contiguos aunque más separados. Ligera mejora.

Doble dispersión ('double hashing')

En doble dispersión: $f(j) = j \cdot d(k)$.

Se usa una nueva función de dispersión 'd' que no devuelve 0. Informalmente: se dispersan los 'clusters'.

Elección típica: $d(k) = q - (k \bmod q)$ con $q < N$ y primo, de forma que el rango de $d(k)$ es el conjunto $\{1, \dots, q\}$.

ERRORES: [Transparencia 14], donde se lee ' d_2 ' debe leerse 'd'.

Complejidad de los métodos de sondeo

El factor de carga $\lambda = n/N$ ahora es menor que 1: como máximo hay una entrada en una posición de la tabla. El tiempo medio del sondeo asumiendo una dispersión uniforme es

$1/(1-\lambda)$ [CLRS'01, p241].

9.2.7 Load Factors and Rehashing

Esta sección aparece en el libro después de una implementación que utiliza redispersión ('rehashing') (Ver [9.2.6 A Java Hash Table Implementation](#)).

Tanto en encadenamiento como en direccionamiento abierto, si el factor de carga no es menor que 1 la tabla no es eficiente.

Recomendaciones [Libro, p401]:

- Encadenamiento: $\lambda < 0.5$.
- Direccionamiento abierto: $\lambda < 0.9$

Redispersión ('rehashing'): cuando el factor de carga no es adecuado,

- Se aumenta el tamaño de la tabla, al menos el doble.
- Se modifican parámetros de la función de dispersión. (N ha cambiado, puede ser necesario recomputar primos como 'q', etc.)
- Se reinsertan todas las entradas en la tabla.

Empeora la eficiencia en el momento de realizarlo, pero mejora la eficiencia de las operaciones a partir de ese momento: buen coste amortizado.

9.2.6 A Java Hash Table Implementation

- **Código:**
 - InvalidKeyException.java
 - HashMap.java.

Implementación de direccionamiento abierto con sondeo lineal. La clase `HashTable<K,V>` implementa el interfaz `Map<K,V>`. La clase anidada `HashTableEntry<K,V>` implementa el interfaz `Entry<K,V>`.

Atributos:

- 'A', el array o tabla.
- 'N', la capacidad (tamaño) del array.
- 'n', el número de entradas en la tabla (el tamaño del TAD).

Métodos:

- 'hashValue', la función de dispersión. Usa el método predefinido en Java 'hashCode' de la clase K y el [Método MAD](#).
- 'findEntry', método auxiliar protegido que implementa el sondeo. Lo invocan 'put', 'get' y 'remove'.
- 'checkKey', sólo comprueba si el objeto clave es 'null'.

- 'rehash', dobla el tamaño del array y recomputa los parámetros de la función de dispersión.

Algunas particularidades:

- Se marca una posición del array como disponible almacenando en ella un objeto 'AVAILABLE' que no es más que una entrada `HashEntry<K,V>` con claves y valor ambos 'null'.
- Array genéricos creados con 'new':
 - o Línea 61: `bucket = (Entry<K,V>[]) new Entry[capacity];`
 - o Línea 137: `bucket = (Entry<K,V>[]) new Entry[capacity];`
- El método `HashEntry<K,V>.equals()` comprueba que dos entradas son iguales comprobando que tienen claves y valores iguales. Este método no es relevante para la tabla que sólo necesita comprobar la igualdad entre claves.
- Código desestructurado. Dos ejemplos:
 - o `findEntry()`: 'break' y 'return' en mitad de un bucle.
 - o `put()`: if-then con return sólo en una rama.

Viernes 15-04-2011

9.5 Dictionaries

Material

- Libro, Sección 9.5.
- Transparencias: [dictionaries.pdf](#).
- Código:
 - o `MultiMap.java` (interfaz)
 - o `HashTableMultiMap.java` (clase).

ERRORES: Estos ficheros no forman parte del paquete `net.datastructures`. Están disponibles en el Aula Virtual (y en [Ch09-fragments](#)).

ERRORES: El interfaz `java.util.Dictionary` de la JCF y el fichero `Dictionary.java` del libro (paquete `net.datastructures`) están obsoletos y no se usan [Libro, p421].

Motivación de los diccionarios

Un diccionario es el grafo (extensión) de una **relación finita**: colección finita de entradas (pares clave-valor). Se permiten múltiples claves iguales.

¿Se permiten **entradas** repetidas (múltiples claves y valores iguales)? El libro no dice nada pero es una pregunta importante cuya respuesta afecta la implementación de los métodos.

Una función finita (Ver [9.1 Maps](#)) es un diccionario que tiene la propiedad funcional.

ERRORES: [Libro, p420] "a dictionary allows for keys and values to be of any object type". Esto contradice el [Requisito mínimo sobre K](#). Recuérdese que el libro también comete el mismo error con las funciones finitas (Ver [Definición matemática de función finita](#)).

Ejemplo epónimo: diccionario de palabras: la clave es la palabra y el valor su acepción. La misma palabra puede tener varias acepciones. Se permiten claves repetidas.

Una tabla en una base de datos es una relación (diccionario en la terminología del libro).

9.5.1 The Dictionary ADT

- **Código:** MultiMap.java.
- [Transparencia 2]

Se define el interfaz MultiMap<K,V>. Métodos de inserción ('put'), búsqueda ('get', 'getAll') y borrado ('remove').

Algunas diferencias con el Map<K,V> del libro (Ver [9.1.1 The Map ADT](#)):

- El método 'put' devuelve la entrada insertada.
- Se usa el interfaz java.util.Map.Entry<K,V> de la JCF (Ver [Maps en la JCF](#)), no el [Interfaz Entry<K,V>](#) del libro (del paquete net.datastructures).
- El método 'get' debe devolver una entrada acorde a algún criterio fijo, por ejemplo la primera entrada encontrada.
- El método 'getAll' debe devolver un iterable de entradas.
- El método 'remove' toma como argumento una entrada (no una clave y un valor por separado). Debe borrar una ocurrencia de la entrada acorde a un criterio fijo.
- Se tiene 'entrySet' pero no 'keySet' y 'values'.
- No se usa 'InvalidKeyException' sino 'java.lang.IllegalArgumentException'.

ERRORES:

- [Libro, p421]: "operation get(k) returns an **arbitrary** entry".
- [Transparencia 2]: "get(k): if the dictionary has an entry with key k, returns it, else, returns null".

Como hemos señalado, 'get' debe devolver una entrada acorde a un criterio fijo, no arbitrario. De otro modo el método 'get' no se comportará como una función: dado un diccionario 'D' que contiene más de una entrada con clave 'k', la invocación 'D.get(k)' debe devolver siempre la misma entrada.

Con 'remove' pasa algo similar si se permiten entradas iguales repetidas (Ver [Motivación de los diccionarios](#)): 'remove' debe o bien borrar todas las copias de la entrada o bien borrar una de ellas según un criterio fijo, por ejemplo la primera encontrada.

- [Libro, p420] y [Transparencia 2] dicen 'iterable collection'. Debe entenderse 'collection' informalmente como iterable, no como Collection<E>.

Un diccionario se puede implementar **ineficientemente** mediante una lista desordenada:

get	$O(n)$
getAll	$O(n)$
remove	$O(n)$
put (permitiendo entradas repetidas)	$O(1)$
put (prohibiendo entradas repetidas)	$O(n)$
entrySet	$O(n)$

ERRORES: En [Libro, Sección 9.5.2 'Implementations with Location-Aware Entries', punto 'Unordered list'] se sugiere una optimización: que cada entrada guarde en un atributo su "localización" en la lista. (Esta técnica se discute en [6.4.2 Sequences](#)).

Según el libro "we can maintain the 'location' variable of each entry e to point to e 's position in the...list. This choice allows us to perform $\text{remove}(e)$ as $L.\text{remove}(e.\text{location}())$, which would run in $O(1)$ time".

- Para `NodePositionList` la localización será la posición, es decir, el nodo que implementa `Position<E>`. En este caso 'remove' y 'put' son $O(1)$.
- Para `ArrayIndexList` la localización será el índice. En este caso, cuando 'remove' borra una entrada se tienen que actualizar los índices de las demás entradas. Para que la complejidad de 'remove' sea $O(1)$ dicha actualización se puede postponer a la siguiente inserción o búsqueda, aumentando la complejidad de éstas. Si la realiza 'remove', entonces su complejidad sigue siendo $O(n)$.

Este es un error del libro, [Transparencias 4-8] son correctas.

9.5.3 An Implementation Using the `java.util` Package

Código: `HashTableMultiMap.java`.

Implementación mediante tabla de dispersión con encadenamiento usando la JCF:

- Interfaz `java.util.Map<K,V>`.
- Interfaz `java.util.Map.Entry<K,V>`.
- Clase `java.util.AbstractMap<K,V>.SimpleEntry<K,V>` que implementa el interfaz `java.util.Map.Entry<K,V>`.
- Clase `java.util.HashMap<K,V>` que es una implementación de `java.util.Map<K,V>` mediante tabla de dispersión.
- Clase `java.util.LinkedList<K,V>` que es una implementación de listas enlazadas.

En vez de almacenar entradas `java.util.Map.Entry<K,V>` de forma similar a la implementación [9.2.6 A Java Hash Table Implementation](#), ahora se almacenan entradas `Map<K,LinkedList<Map.Entry<K,V>>>`: el valor asociado a una clave es la lista (implementada por la clase `java.util.LinkedList<T>`) de entradas con la misma clave.

En abstracto, en vez de tener una relación finita donde hay entradas con claves repetidas:

$$\{ (k_1, v_1), (k_2, v_2), (k_3, v_3), (k_1, w_1), (k_3, w_3) \}$$

Se tiene una relación finita de entradas en las que los valores son la lista de entradas (indicada por los corchetes) con la misma clave:

$$\{ (k_1, [(k_1, v_1), (k_1, w_1)]), (k_2, [(k_2, v_2)]), (k_3, [(k_3, v_3), (k_3, w_3)]) \}$$

No habrá entradas con listas vacías, p.ej. $(k_4, [])$. Tampoco hay entradas (pares clave+lista-de-entradas) con claves repetidas. Pero puede haber entradas repetidas en la lista-de-entradas.

Dos claves k_1 y k_2 colisionan si $k_1=k_2$ o si $k_1 \neq k_2$ pero $h(k_1)=h(k_2)$.

Los métodos 'put', 'get', 'getAll' y 'remove' comprueban si las claves o las entradas (y las claves de las entradas) pasadas como argumento son 'null'. Lanza 'IllegalArgumentException' en caso positivo.

El método 'remove' de la tabla delega el borrado en el método 'remove' de `LinkedList` [Código, línea 57]. La clase `LinkedList` define varios métodos 'remove'. El método que se invoca es 'boolean remove(Object o)' que según la documentación Javadoc "removes the first occurrence of the specified element from this list, if it is present". Si la lista queda vacía se borra la entrada en la tabla [Código, línea 59].

Ejemplos con el diccionario en la línea 3469:

- `remove((k2, v2))` elimina la entrada (k_2, v_2) de la lista en la entrada $(k_2, [(k_2, v_2)])$. El resultado es la entrada $(k_2, [])$ que se borra de la tabla al ser su lista vacía. La tabla queda sin entradas con clave k_2 .
- seguido de `remove((k1, w1))`:

$$\{ (k_1, [(k_1, v_1)]), (k_3, [(k_3, v_3), (k_3, w_3)]) \}$$

El método 'put' de la tabla delega en el método 'put' de `HashMap` y en el método 'add' de `LinkedList`. En [Código, línea 20] no se está preguntando si la entrada tiene una lista vacía (cosa que no sucederá nunca). Se está preguntando si la búsqueda de la clave en el objeto 'm' de clase `HashMap` ha devuelto un valor (una lista) o null (la clave no está en 'm').

Ejemplos con el diccionario en la línea 3469:

- `put((k1, w2))`:
- $$\{ (k_1, [(k_1, v_1), (k_1, w_1), (k_1, w_2)]), (k_2, [(k_2, v_2)]), \dots \}$$
- seguido de `put((k1, v1))`:

$$\{ (k_1, [(k_1, v_1), (k_1, w_1), (k_1, w_2), (k_1, v_1)]), (k_2, [(k_2, v_2)]), \dots \}$$

El método 'get' delega en el método 'peekFirst' de LinkedList que según la documentación Javadoc "Retrieves, but does not remove, the first element of this list, or returns null if this list is empty".

Ejercicio: Implementar el método 'put' de forma que no se almacenen entradas repetidas (Ver [Motivación de los diccionarios](#)).

El método 'getAll' devuelve la lista de entradas asociada a la clave.

El método 'entrySet' devuelve un objeto LinkedList (que es iterable) con todas las entradas. Para ello se usan los métodos 'values' de HashMap y 'addAll' de LinkedList.

Viernes 06-05-2011

9.3 Ordered Maps

Material

Libro: Sección 9.3.

Transparencias: binarysearchtrees.pdf (transparencias 2-4).

No hay código asociado.

Motivación de las funciones finitas con dominio ordenado

En el libro, '9.3 Ordered Maps' viene antes que '9.5 Dictionaries'. En este guión viene después porque las funciones finitas con dominio ordenado están relacionadas con los árboles de búsqueda binarios que veremos en el siguiente tema.

Funciones finitas con dominio ordenado: son **funciones finitas** (no hay claves repetidas) **con una relación de orden total sobre el dominio** (sobre el conjunto de claves). El objetivo de tener claves con orden es realizar operaciones (inserción, búsqueda, borrado) basándose en dicho orden.

Ya hemos visto:

- Funciones finitas: [9.1 Maps](#).
- Órdenes totales: [8.1.1 Keys, Priorities, and Total Order Relations](#).
- Realización de órdenes totales en Java: [8.2.1 Entries and Comparators](#).

Interfaz [Transparencia 2, binarysearchtrees.pdf] [Libro, p403]: extiende el interfaz de funciones finitas [9.1.1 The Map ADT](#) y añade los métodos 'firstEntry', 'lastEntry', 'ceilingEntry', 'floorEntry', 'lowerEntry' y 'higherEntry', las dos últimas no mencionadas en [Transparencia 2]:

- lowerEntry(k) devuelve la mayor clave k' tal que k' < k, o null.

- `higherEntry(k)` devuelve la menor clave k' tal que $k < k'$, o null.

ERRORES: En [Transparencia 2] "`firstEntry()` entry with smallest key value " en vez de "`firstEntry()` entry with smallest key or null if map empty". También dice "Keys are assumed to come from a total order" lo cual es impreciso.

En la JCF se llaman "funciones finitas navegables" (interfaz java.util.NavigableMap). Navegable puede entenderse como que se puede iterar la función finita siguiendo el orden de las claves.

Ejemplos de funciones finitas con dominio ordenado:

1. Reservas de vuelos [Libro, p408]: La clase de claves tiene atributos origen, destino, fecha y hora. La claves ordenadas lexicográficamente. La clase de los valores tiene atributos duracion del vuelo, precio, número del vuelo, asiento, etc.
Los clientes quieren que la búsqueda les devuelva, para el mismo origen y destino, una lista de vuelos en distintas fechas y horas. Se pueden usar métodos los '`floorEntry`' y '`ceilingEntry`' para obtener dichas listas.
2. Listín telefónico: al estar ordenado alfabéticamente se facilita la búsqueda.

Ejercicio: *¿De los TADs e implementaciones vistas hasta el momento, cuales sí y cuáles no tendría sentido utilizar para implementar funciones finitas con orden?*

9.3.1 Ordered Search Tables and Binary Search

Posible implementación: `ArrayIndexList` que almacena las entradas ordenadas por clave:

$[(k_1, v_1), \dots, (k_n, v_n)]$ con $k_1 < \dots < k_n$.

Factor esencial: obtener la entrada en el índice i es $O(1)$.

¿Por qué no hemos escrito $k_1 \leq \dots \leq k_n$?

Complejidad de las operaciones:

- **La búsqueda es $O(\log n)$:** '`get`', '`floorEntry`', '`ceilingEntry`', '`firstEntry`', '`lastEntry`', '`lowerEntry`' y '`higherEntry`'. Se realiza mediante búsqueda binaria descrita abajo.
- **La inserción y borrado son $O(n)$:** búsqueda binaria mas despazamiento a izquierda ('`remove`') o a derechas ('`put`') de entradas en el vector.

Búsqueda binaria [Transparencia 3]. Búsqueda binaria en Java adaptada del pseudocódigo en [Libro, p405]:

```
#+BEGIN_EXAMPLE
```

```
public Entry<K,V> binarySearch(IndexList S, K k, Comparator<K> c, int
low, int high) {
```

```
    int mid,r ;
```

```
    Entry<K,V> e ;
```

```

if (low > high) return null ;

else {

    mid = (low + high) / 2 ;

    e    = S.get(mid) ;

    r    = c.compare(k,e.getKey()) ;

    if (r == 0)

        return e ;

    else if (r < 0)

        return binarySearch(S, k, low, mid-1) ;

    else

        return binarySearch(S, k, mid+1, high) ;

}

}

```

#+END_EXAMPLE

Búsqueda "binaria": sólo hay dos posibilidades de búsqueda cuando la clave buscada no está en la entrada almacenada en 'S.get(mid)'. En cada llamada se reduce a la mitad el rango de la búsqueda: $O(\log n)$.

Eliminación de la recursión de cola ('tail recursion') por el compilador:

#+BEGIN_EXAMPLE

START:

```

if (low > high) return null ;

else {

    mid = (low + high) / 2 ;

    e    = S.get(mid) ;

    r    = c.compare(k,e.getKey()) ;

    if (r == 0)

        return e ;

    else if (r < 0) {

        high = mid-1 ;

        goto START ;

    } else {

        low = mid+1 ;

    }

}

```

```
goto START ;
```

```
}
```

```
}
```

```
#+END_EXAMPLE
```

ERRORES [Transparencia 4]:

- En el segundo punto de "Performance" dice "get takes $O(n)$ " cuando debería de decir "put takes $O(n)$ ".
- En los dos últimos puntos de "Performance", donde dice " $n/2$ " debería decir " n ", pues el elemento a insertar o eliminar puede ser el primero.

Tabla comparativa para varias implementaciones [Libro, p407]:

Método	IndexList	PositionList	HashMap
get (y otros)	$O(\log n)$	$O(n)$	Esp: $O(1)$ Peor: $O(n)$
put	$O(n)$	$O(1)$	$O(1)$
remove	$O(n)$	$O(n)$	Esp: $O(1)$ Peor: $O(n)$
entrySet	$O(n)$	$O(n)$	$O(n)$

Ejercicio: Definir el interfaz '*OrderedMap<K,V>*' e implementarlo usando '*ArrayIndexList<Entry<K,V>>*'.

Los [10.1 Binary Search Trees](#) son una forma de implementar funciones finitas con dominio ordenado en el que todas las operaciones son $O(\log n)$.

10.1 Binary Search Trees

Material

Libro: Sección 10.1.

Transparencias: binarysearchtrees.pdf (transparencias 5-10).

Código:

- BinarySearchTreeMap.java.
- BinarySearchTree.java. <-- **obsoleto**
- Carpeta BinTreeBinSearchTree (código explicativo sobre aspectos de implementación de los árboles binarios y los árboles binarios de búsqueda).

ERRORES: [Libro, Sección 10.1.3] El texto dice 'BinarySearchTree' cuando debería decir 'BinarySearchTreeMap'.

El fichero BinarySearchTree.java implementa diccionarios definidos mediante el interfaz 'Dictionary' que está obsoleto (véase la sección 'Material' de [9.5 Dictionaries](#)).

Repasar: [7 Tree Structures](#), en especial [7.3 Binary Trees](#).

Motivación de los árboles binarios de búsqueda.

Los árboles son una representación idónea para funciones finitas y diccionarios con dominio ordenado. En este tema presentamos los árboles binarios llamados "de búsqueda", que permiten realizar búsqueda binaria, inserción y borrado en $O(h)$ donde h es la altura del árbol.

Notación en libro y transparencias: se usan 'v', 'w', etc, para nodos del árbol, no para valores.

Definición de árbol binario de búsqueda:

Son árboles binarios con las siguientes características:

- Para representar **funciones finitas** con dominio ordenado: para todo nodo interno con entrada (k, x) , las claves en el subárbol izquierdo del nodo serán **menores** que k y las claves en el subárbol derecho serán **mayores** que k .
- Para representar **diccionarios** con dominio ordenado: para todo nodo interno con entrada (k, v) , las claves en el subárbol izquierdo del nodo serán **menores o iguales** que k y las claves en el subárbol derecho serán **mayores** que k . (Otra posibilidad es elegir el subárbol derecho para almacenar las entradas mayores o iguales a k , quedando en el subárbol izquierdo sólo las menores que k .) Otras posibilidades para almacenar entradas con claves repetidas que no trataremos aquí. Por ejemplo:
 - Alternar entre los subárboles izquierdo y derecho siguiendo un criterio fijo que permita realizar búsqueda binaria.
 - Usar una colección para almacenar las entradas con claves repetidas en un único nodo.

ERRORES: [Libro, p432]: La definición de árbol binario de búsqueda en esa página del libro es para representar diccionarios, no funciones finitas que es de lo que en realidad trata la sección del libro. Además se debe especificar claramente en qué hijo están localizadas las claves iguales. No pueden estar en los dos subárboles hijos a la vez a no ser que sea acorde a un criterio que permita realizar búsqueda binaria.

Representación mediante árboles binarios

El libro usará árboles binarios [7.3 Binary Trees](#) con la siguiente particularidad de representación: **los árboles binarios de búsqueda vacíos y los nodos externos serán representados por nodos con elemento a null.**

En un árbol binario los nodos externos son (por definición) nodos que no tienen hijos. No se dice que tengan que almacenar elementos no null. De hecho, el método 'checkPosition' de la clase [Clase LinkedBinaryTree<E>](#) no realiza (afortunadamente) esta comprobación.

En las figuras de árboles binarios de búsqueda del libro y las transparencias los nodos externos se dibujan como cuadrados vacíos. (Repasar lo dicho en [Cuestiones de implementación de árboles generales](#)).

El motivo de usar esta representación está explicado en [Libro, p432]: "we store entries only at the internal nodes of a binary search tree, and external nodes serve as "placeholders. [...] we could have allowed for improper binary trees, which have a better space usage, but at the expense of more complicated search and update methods".

Realmente será la inserción de nodos (método 'insertAtExternal') lo que resulta más fácil de implementar con esta representación. Sin embargo, cuestionaremos en [Críticas a la implementación BinarySearchTreeMap](#) la verdad de ésta afirmación.

Definiciones:

- **Nodo frontera:** nodo interno cuyos hijos son nodos externos (sus hijos son nodos con elemento a null). En un árbol binario, los nodos frontera serían los nodos externos (su elemento no es null y no tienen hijos).
- **Frontera de un árbol:** conjunto de sus nodos frontera.

Consecuencias:

Un árbol binario que consiste en un único nodo con elemento null no es un árbol binario vacío pero sí un árbol binario de búsqueda vacío.

[Libro, p432]: "we can take the view that binary search trees are nonempty proper binary trees". Son árboles propios pues todo nodo interno tiene dos hijos: en el caso de los nodos frontera, dos hojas (nodos con elemento null).

En el código de la [Clase LinkedBinaryTree<E>](#) se han tomado decisiones para acomodar esta representación. (Véanse los comentarios iniciales en el código y [10.1.3 Java Implementation](#) más abajo.)

10.1.1 Searching

[Transparencia 6]

El pseudocódigo `TreeSearch` realiza la búsqueda binaria en el árbol. Obsérvese que `TreeSearch(v)` devuelve el nodo `v` cuando es externo y la clave no está en el árbol. También devuelve el nodo `v` cuando la clave está en dicho nodo.

El método 'get' debe invocar 'TreeSearch' sobre el nodo raíz. Debe devolver null si el nodo devuelto por 'TreeSearch' es externo y devolver el valor en la entrada (elemento) del nodo cuando éste es interno:

```
#+BEGIN_EXAMPLE

    Algorithm get(k):
        v ← TreeSearch(k, T.root())
```

```

        if T.isExternal(v) then
            return null
        else
            return v.element().getValue()

#+END_EXAMPLE

```

En el caso de diccionarios el pseudocódigo devuelve el valor de la primera entrada encontrada con clave k o null.

10.1.2 Update Operations

Insertion

[Transparencia 7]

Método 'put(k,x)', usamos ' x ' para el valor pues ' v ' se usa para nodos.

La inserción es diferente para funciones finitas y diccionarios: si la clave está en el árbol, para funciones finitas se actualiza la entrada mientras que para diccionarios se inserta la nueva entrada como si la clave no estuviera en el árbol. Se asume un método auxiliar 'insertAtExternal(v,e)' que convierte el nodo externo ' v ' en interno con elemento (entrada) ' e '.

ERRORES: [Libro p435, Code Fragment 10.2]. El pseudocódigo de 'TreeInsert' permite insertar entradas con claves repetidas, que sólo es aplicable a diccionarios. Además, para funciones finitas el método 'put' devuelve el valor antiguo si la clave ya está en la función [9.1.1 The Map ADT](#) y para diccionarios devuelve la entrada insertada [9.5.1 The Dictionary ADT](#).

A continuación mostramos versiones para funciones finitas y diccionarios.

Pseudocódigo de 'put' (común a funciones finitas y diccionarios):

```

#+BEGIN_EXAMPLE

    Algorithm put( $k, x$ ):
        return TreeInsert( $k, x, T.root()$ )

#+END_EXAMPLE

```

Pseudocódigo 'TreeInsert' para funciones finitas:

```

#+BEGIN_EXAMPLE

    Algorithm TreeInsert( $k, x, v$ )
         $w \leftarrow$  TreeSearch( $k, v$ )
        if T.isInternal( $w$ ) then

```

```

                                old = w.element().getValue() // Guarda el valor
antiguo.
                                T.replaceEntry(w, (k,x))          // Actualiza
la entrada.
                                return old                      // Devuelve el
valor antiguo.
                                else
                                T.insertAtExternal(w, (k,x))
                                return null
#+END_EXAMPLE

```

Pseudocódigo de 'TreeInsert' para diccionarios:

```

#+BEGIN_EXAMPLE
Algorithm TreeInsert(k,x,v)
w <- treeSearch(k,v)
if T.isInternal(w) then
// Se inserta en cualquiera de los hijos, p.ej, el izdo.
    TreeInsert(k,x,T.left(w))
else
    T.insertAtExternal(w, (k,x))
    return (k,x) // Devuelve la entrada insertada
#+END_EXAMPLE

```

Borrado

[Transparencias 8-9]

Método 'remove(k)' para funciones finitas y 'remove(e)' para diccionarios. (Véase [9.1.1 The Map ADT](#) y [9.5.1 The Dictionary ADT](#).)

- Para funciones finitas se elimina la entrada con clave k.
- Para diccionarios se podría eliminar o bien la primera ocurrencia encontrada de la entrada o bien todas las ocurrencias de la entrada.

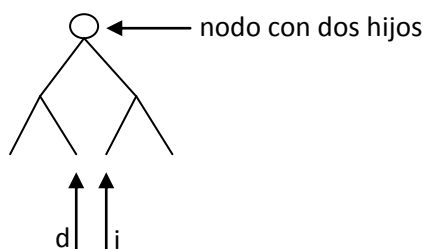
La entrada con clave k puede estar en un nodo interno:

1. Sin hijos.
2. Con un único hijo.
3. Con dos hijos.

Para 1-2 se utiliza un método auxiliar 'removeExternal(v)' que borra el nodo externo 'v' del árbol y además reemplaza al padre (el nodo interno donde está la clave) de 'v' con el hermano de 'v' (que puede ser un nodo externo).

Para 3, se mueve al nodo donde está la clave buscada la entrada de alguno de éstos nodos:

- A. El nodo interno 'd' más a la derecha del hijo izquierdo.
- B. El nodo interno 'i' más a la izquierda del hijo derecho.



y posteriormente se invoca 'removeExternal' sobre el hijo derecho de 'd' (que es un nodo externo) o el hijo izquierdo de 'i' (que es un nodo externo), borrándose las hojas y el padre ('d' o 'i').

[ALT'03, p419]: En el **borrado asimétrico** la elección del nodo a mover está fija (se elige A siempre o B siempre). Así el borrado afecta siempre al mismo subárbol. En el **borrado asimétrico** se alterna entre A y B según algún criterio, fijo o aleatorio.

Ejercicio: ¿Cómo sería el borrado para diccionarios?

Complejidad para árboles binarios de búsqueda

Complejidad de los métodos:

get (y otros)	$O(h)$
Put	$O(h)$
remove	$O(h)$
entrySet	$O(h)$

Caso **peor**, "árbol degenerado": $h = n$.

Caso **árbol binario equilibrado**: $h = \log n$. Al estar equilibrado, cuando el número de nodos crece exponencialmente (en base 2) la altura crece logarítmicamente.

Problema: garantizar el equilibrado.

En un árbol binario ordinario la búsqueda es $O(n)$ pues tendría que realizarse mediante el iterador: métodos 'iterator' o 'positions'.

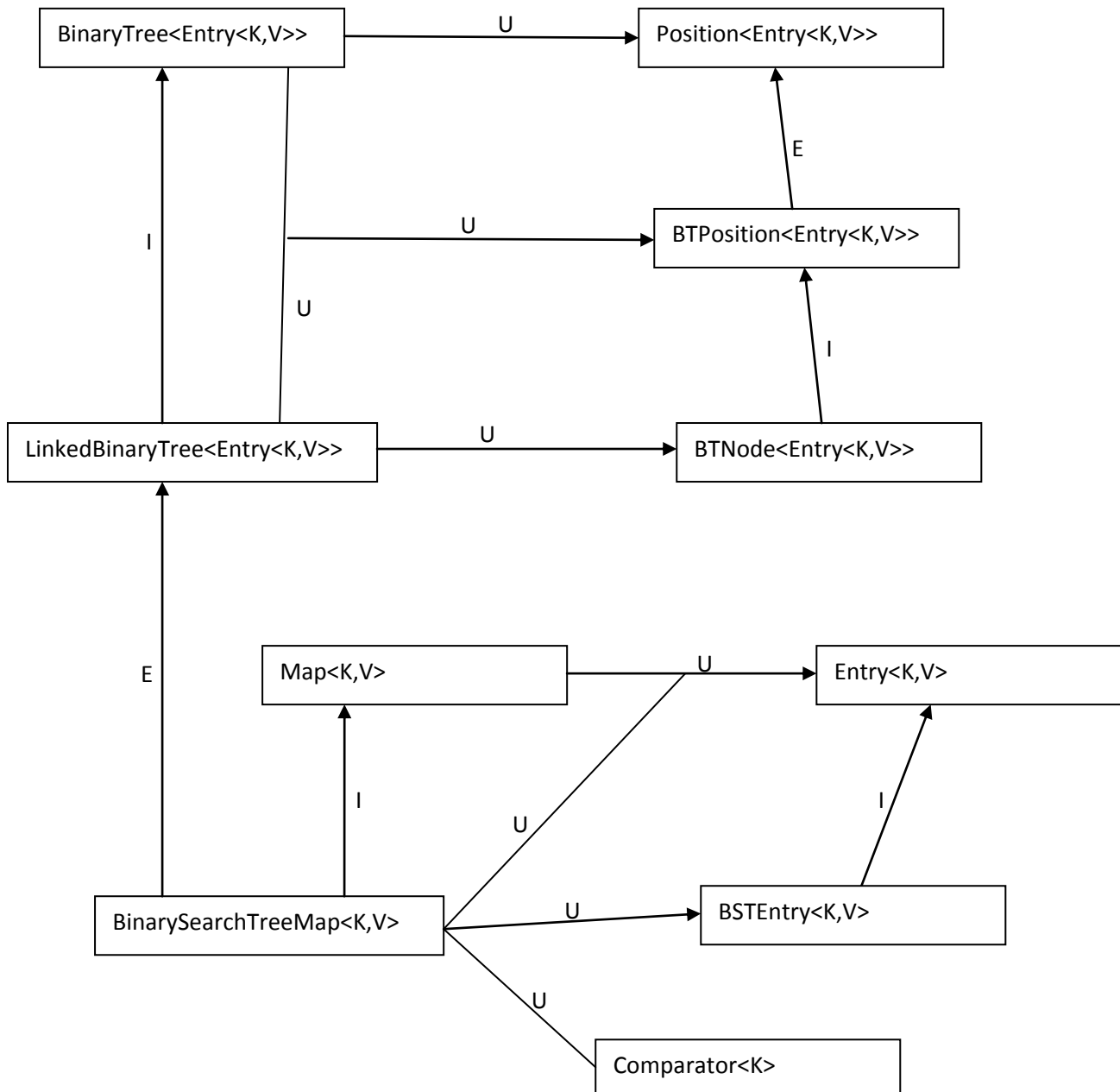
10.1.3 Java Implementation

Código: BinarySearchTreeMap.java.

La clase 'BinarySearchTreeMap<K,V>' extiende la [Clase LinkedBinaryTree<E>](#) (que implementa árboles binarios) e implementa el interfaz 'Map<K,V>' (funciones finitas [9.1.1 The Map ADT](#)). Se asume dominio ordenado y la [Representación mediante árboles binarios](#).

Diagrama de clases involucradas:

#+BEGIN_EXAMPLE



#+END_EXAMPLE

- U: Usa. I: Implementa. E: Extiende.

Obsérvese que no se extiende la [Clase BTreeNode<E>](#): los nodos del árbol binario de búsqueda son nodos de un árbol binario. Se definen tres atributos: un comparador de claves 'C', un nodo especial 'actionPos' y el número de entradas en el árbol 'numEntries'. Recuérdese que el atributo 'size' heredado de la [Clase LinkedBinaryTree<E>](#) almacena el número total de nodos del árbol (incluyendo, por lo tanto, nodos externos con elemento a null del árbol binario de búsqueda).

El nodo 'actionPos' (atributo protegido) es usado por los métodos:

- 'get', para almacenar el nodo en el que terminó la búsqueda.
- 'put', para almacenar el nodo en el que se inserta la entrada.
- 'remove', para almacenar el hermano del nodo externo borrado.

No es usado para nada más en esta clase pero es usado en futuras clases hijas, como veremos en [10.2 AVL Trees](#). Otro caso cuestionable de una clase padre cuya implementación está determinada por clases hijas.

Ahora podemos explicar los comentarios en la [Clase LinkedBinaryTree<E>](#) sobre el atributo 'size'. En esa clase el método 'size' devuelve el número total de nodos del árbol. En la Clase `BinarySearchTreeMap<K,V>`, que implementará árboles binarios de búsqueda, el método 'size' debe devolver el número total de entradas almacenadas, ignorando los nodos externos con elemento a null.

La clase 'BinarySearchTreeMap' sobrescribe el método 'size':

- El método 'checkPosition' de 'LinkedBinaryTree' considera como válidos nodos con elemento a null.
- Un objeto de clase 'BinarySearchTreeMap' puede invocar métodos heredados de 'LinkedBinaryTree', algunos de los cuales ('isEmpty', 'positions', 'attach') necesitan conocer el tamaño del árbol binario. Si obtienen el tamaño a través del método 'size', **debido al enlazado dinámico** obtendrán el valor devuelto por el método 'size' de 'BinarySearchTreeMap', es decir, el tamaño del árbol como árbol binario de búsqueda, no como árbol binario. Para evitarlo, los métodos de 'LinkedBinaryTree' obtienen el tamaño directamente del atributo 'size', que no se sobrescribe en 'BinarySearchTreeMap', pues se usa el nuevo atributo 'numEntries'.

Como dijimos en [Clase LinkedBinaryTree<E>](#), se trata de un caso cuestionable de una clase padre cuya implementación está determinada por clases hijas.

- Véase el código explicativo en la carpeta `BinTreeBinSearchTree` y la discusión en [Críticas a la implementación BinarySearchTreeMap](#) donde se comenta dicho código explicativo.

Los dos constructores crean un árbol de búsqueda vacío. El primer constructor crea un 'DefaultComparator' y un nodo raíz con elemento a null [Representación mediante árboles binarios](#). El segundo constructor toma el comparador como argumento y crea un nodo raíz con elemento a null. El atributo 'numEntries' podría haber sido inicializado por los constructores en vez de en la declaración [Código, línea 20].

La clase anidada 'BSTEntry<K,V>' implementa el [Interfaz Entry<K,V>](#). Los objetos de esta clase son las entradas que se almacenan en el árbol.

- Atributos: la clave 'key', el valor 'value' y la posición (nodo del árbol) 'pos' donde se almacenada esta entrada. En el código de 'BinarySearchTreeMap' se accede directamente al atributo 'pos' para almacenar el nodo, no se usa un "getter". Obsérvese la circularidad: una entrada guarda una referencia al nodo donde está almacenada y un nodo guarda una referencia a la entrada que almacena.
- Constructores: constructor por defecto (todos los atributos a null) y constructor que toma valores para los tres atributos.
- Métodos: "getters" para los atributos.

Se añaden métodos protegidos "getters" ('key', 'value', 'entry') para obtener la clave, el valor y la entrada almacenada en un nodo del árbol binario de búsqueda. Se añaden simplemente para abreviar, pues pueden obtenerse usando el método 'element' del nodo (interfaz 'Position') tal y como hace el código de dichos "getters".

Se añade el método protegido 'replaceEntry', un "setter" que reemplaza la entrada almacenada en un nodo por otra y devuelve el valor en la entrada antigua. El método hace casting del nodo a 'BSTEntry' (pues recibe un Entry<K,V> como argumento) e invoca el método 'replace' heredado de la clase 'LinkedBinaryTree'. El método 'replaceEntry' es la implementación del método tocayo mencionado en el pseudocódigo visto en [10.1.2 Update Operations](#).

ERRORES: [Código, línea 64, método 'replaceEntry'] El casting podría lanzar una excepción no recogida si la entrada no es de clase 'BSTEntry'. El método debería invocar 'checkEntry' antes del casting.

Los métodos 'checkKey' y 'checkEntry' comprueban respectivamente la validez de una clave (que no sea null) y de una entrada (que no sea null y que sea una instancia de 'BSTEntry').

El método esencial 'insertAtExternal' invoca el método 'expandExternal' heredado de 'LinkedBinaryTree' y el "setter" 'replace' descrito arriba. Actualiza el número de entradas.

El método 'removeExternal' invoca el método 'removeAboveExternal' heredado de 'LinkedBinaryTree' y actualiza el número de entradas en el árbol binario de búsqueda. El método 'removeAboveExternal' borra el nodo externo 'v' y reemplaza el padre por su hermano. El nodo 'v' es un nodo externo (con elemento null y dos hijos null) y también lo es su hermano, según la [Representación mediante árboles binarios](#). Su hermano es también un nodo externo. Por tanto, 'removeAboveExternal' deja al padre de 'v' como nodo externo (con elemento null y dos hijos null).

El método 'treeSearch' implementa la búsqueda binaria [10.1.1 Searching](#).

Se sobrescribe el método 'size' que devuelve el número de entradas 'numEntries' y el método 'isEmpty' usa el método 'size' (no el atributo 'size' heredado de 'LinkedBinaryTree'. Ver [Representación mediante árboles binarios](#)).

El código de 'put' es una variación del pseudocódigo visto en la sección [10.1.2 Update Operations](#). Se codifica directamente 'TreeInsert'. Se usa el método protegido 'replaceEntry' cuando la clave está en el árbol.

ERRORES: [Código, línea 142, método 'put'], la invocación a 'getValue()' es redundante y debe omitirse.

El código de 'remove' comprueba los tres casos descritos en la sección [10.1.2 Update Operations](#). En el caso de que la entrada a borrar esté en un nodo con dos hijos que son nodos internos se ejecuta la opción B: se elige para reemplazar al hijo más a la izquierda del subárbol derecho (primero se invoca 'right' y luego se entra en un bucle que invoca 'left' hasta llegar al nodo frontera).

Los métodos 'keySet', 'values' y 'entrySet' devuelven un 'NodePositionList' con las claves, valores y entradas, respectivamente, en el árbol binario de búsqueda. Los tres métodos invocan el método 'positions' heredado de 'LinkedBinaryTree' que devuelve un iterable con todos los nodos del árbol binario. Recorren el iterable guardando en la lista la clave, el valor, o la entrada, respectivamente, almacenada en los nodos **internos** (Ver [Representación mediante árboles binarios](#)).

Los métodos 'swapElements', 'expandExternal' y 'removeAboveExternal' utilizados por los métodos de esta clase son los heredados de la clase 'LinkedBinaryTree'. Se implementa pero no se usa el método 'restructure'. Se trata de un método que veremos en [10.2 AVL Trees](#). El método no debería estar en esta clase. Véase [Críticas a la implementación BinarySearchTreeMap](#).

Críticas a la implementación BinarySearchTreeMap

Hemos cuestionado diversos aspectos de la implementación de árboles binarios y árboles binarios de búsqueda en [Clase LinkedBinaryTree<E>](#), [Representación mediante árboles binarios](#) y [10.1.3 Java Implementation](#).

Uno de los problemas fundamentales es el uso de la herencia como mero mecanismo de reutilización de código, sin cumplirse el Principio de Sustitución de Liskov (LSP) o la relación **es-un**. Un árbol binario de búsqueda no es un árbol binario (no se comporta igual) sino un caso especial, una excepción. La sobrescritura no basta para tratar la excepcionalidad. Las clases padre están polucionadas con código motivado por el funcionamiento de las clases hijas.

El origen es la [Representación mediante árboles binarios](#) en la que los nodos externos son nodos vacíos con elementos a null. El motivo de ésta representación, según el libro, es facilitar la implementación, en particular la inserción de nodos (método 'insertAtExternal'). Nos cuestionamos esta afirmación con la siguiente pregunta: ¿Es más complicada la inserción (y toda la implementación) en una representación como en árboles binarios (con nodos externos que tienen información)? ¿Qué métodos de las clases LinkedBinaryTree y BinarySearchTreeMap habría que modificar y de qué manera?

El método 'restructure' no debería estar en la clase 'BinarySearchTreeMap'. Debería estar en una clase hija que veremos en [10.2 AVL Trees](#).

Viernes 13-05-2011

10.2 AVL Trees

Material

Libro: Sección 10.2

Transparencias: avltrees.pdf.

Código:

- AVLTreeMap.java.
- AVLTree.java. <-- **obsoleto**

ERRORES: [Libro, Sección 10.2.2] El texto dice 'AVLTree' cuando debería decir 'AVLTreeMap'.

ERRORES: El fichero AVLTree.java implementa árboles AVL para diccionarios definidos mediante el interfaz 'Dictionary' que está obsoleto (véanse las secciones 'Material' de [9.5 Dictionaries](#) y de [10.1 Binary Search Trees](#)). Repasar [10.1 Binary Search Trees](#).

Motivación de los árboles AVL

[Transparencias 2-3]

Son **árboles binarios de búsqueda** que se **autoequilibran** (mantienen la propiedad de [árbol binario equilibrado](#), llamada 'Height-Balance Property' en [Libro, p443]) gracias a operaciones denominadas "de rotación" realizadas durante la inserción y el borrado.

En el libro y las transparencias se asume que no hay claves repetidas: los árboles AVL implementarán funciones finitas con dominio ordenado.

Los árboles AVL toman su nombre de las iniciales de los apellidos de sus dos inventores: Adelson-Velskii y Landis.

En [Complejidad para árboles binarios de búsqueda](#) vimos que la complejidad de la inserción, borrado y búsqueda en árboles binarios de búsqueda equilibrados es $O(\log n)$.

Representación mediante árboles binarios de búsqueda

Se utilizan árboles binarios de búsqueda cuyos nodos guardan su altura. En algunas representaciones los nodos guardan un entero llamado **factor de equilibrio** ('balance factor') que indica la diferencia de altura de sus hijos. El factor de equilibrio se puede calcular a partir de la altura. Recuérdese lo dicho en [Representación mediante árboles binarios](#) sobre nodos con elemento null, etc.

10.2.1 Update Operations

Inserción

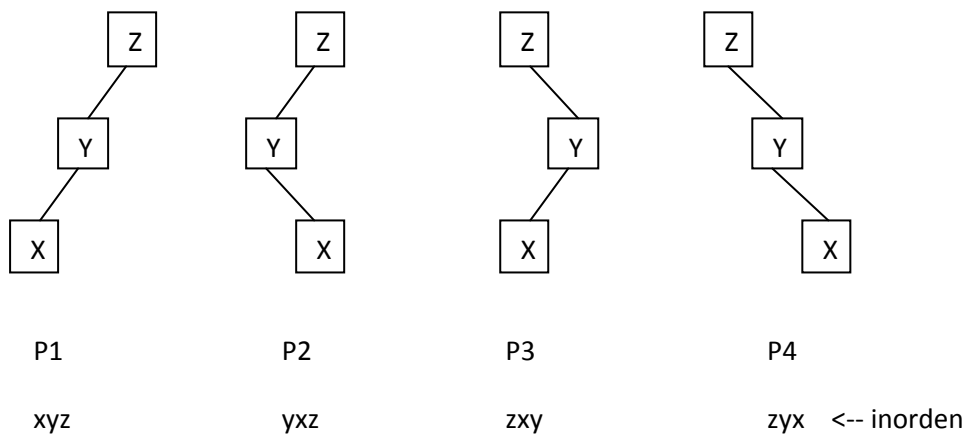
Si la clave está en el árbol se actualiza la entrada en el nodo correspondiente. Si la clave no está en el árbol se inserta la entrada como en un árbol binario de búsqueda (Ver [10.1.2 Update Operations](#) (se crea un nuevo nodo externo).

Se comprueba el equilibrio de los nodos en el camino del nodo insertado a la raíz. Se produce desequilibrio cuando hay un nodo tal que el valor absoluto de la diferencia entre la altura de sus hijos es mayor que 1.

Sea 'w' el nodo insertado. En el desequilibrio siempre están involucrados tres nodos ancestros de 'w' que llamamos 'z', 'y', 'x', donde 'z' es el primer nodo desequilibrado en el camino de 'w' a la raíz, 'y' es el hijo de 'z' de mayor altura y 'x' es el hijo de 'y' de mayor altura.

Posibles combinaciones de 'z', 'y', 'x':

#+BEGIN_EXAMPLE



#+END_EXAMPLE

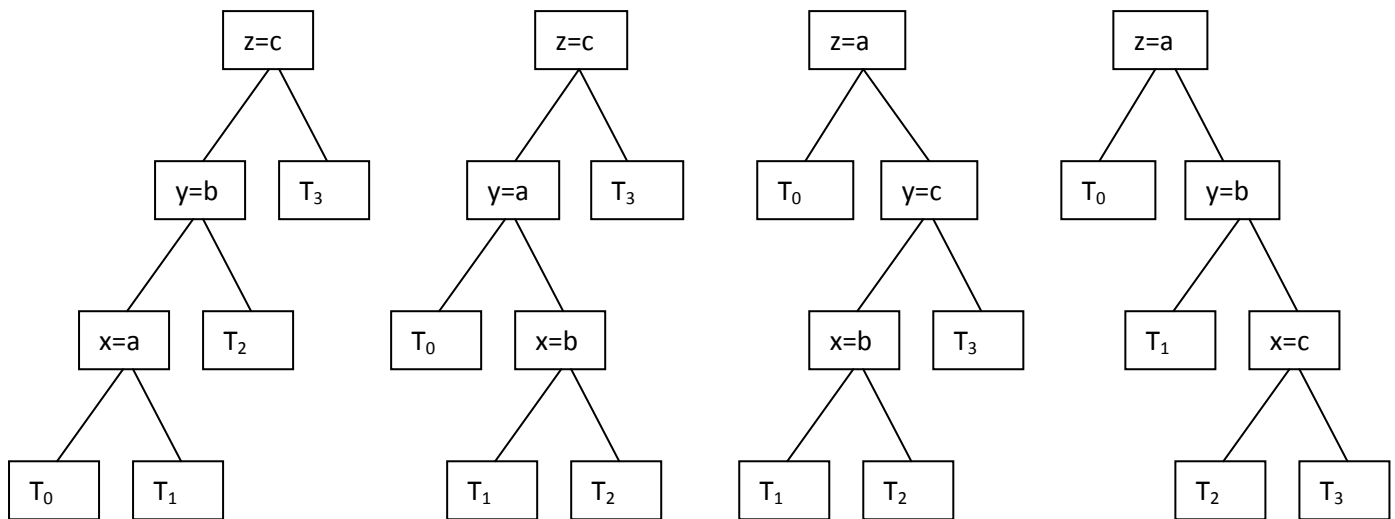
P1 y P4 son rectas. P2 y P3 son cuñas.

El equilibrio debe mantener la propiedad de árbol binario de búsqueda: debe mantenerse el recorrido en inorden.

Algoritmo de equilibrio 'restructure(x)': se aplica sobre el nodo 'x' ('y' es el padre de 'x' y 'z' es el padre de 'y', alcanzables fácilmente). El algoritmo se describe en detalle en [Libro, p447]. A resaltar:

- Se renombran los nodos como 'a', 'b', 'c' según el inorden, de forma que 'b' tiene 'a' como hijo izquierdo y 'c' como hijo derecho.

#+BEGIN_EXAMPLE



P1

P2

P3

P4

xyz

yxz

zxy

zyx

abc

abc

abc

abc

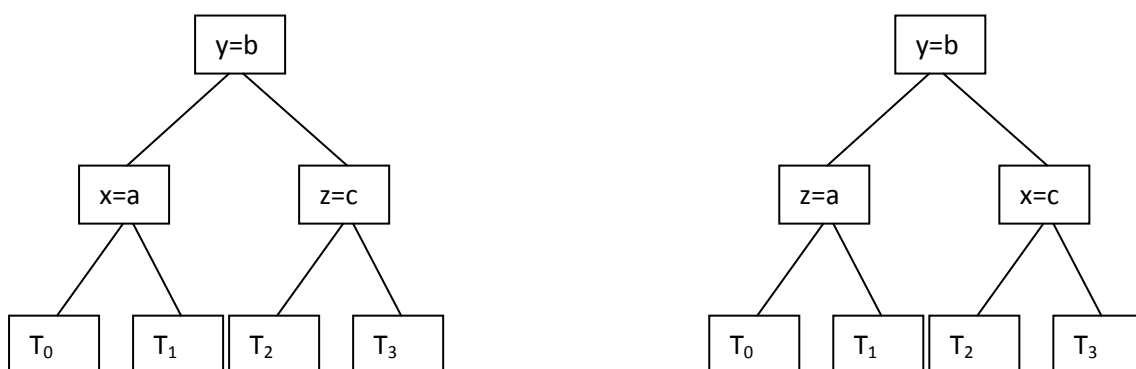
#+END_EXAMPLE

- Hay dos tipos de rotaciones [Libro, p448] [Transparencias 7-8]:

1. Simples ('single rotation'). Cuando $b=y$ (P1 y P4).

Se rota 'y' sobre 'z'. Un hijo de 'y' (T_2 en P1 y T_1 en P4) pasa a ser hijo de 'z' poniéndose 'z' en lugar de dicho hijo de 'y'.

#+BEGIN_EXAMPLE



P1

P4

xyz

zyx <-- inorden

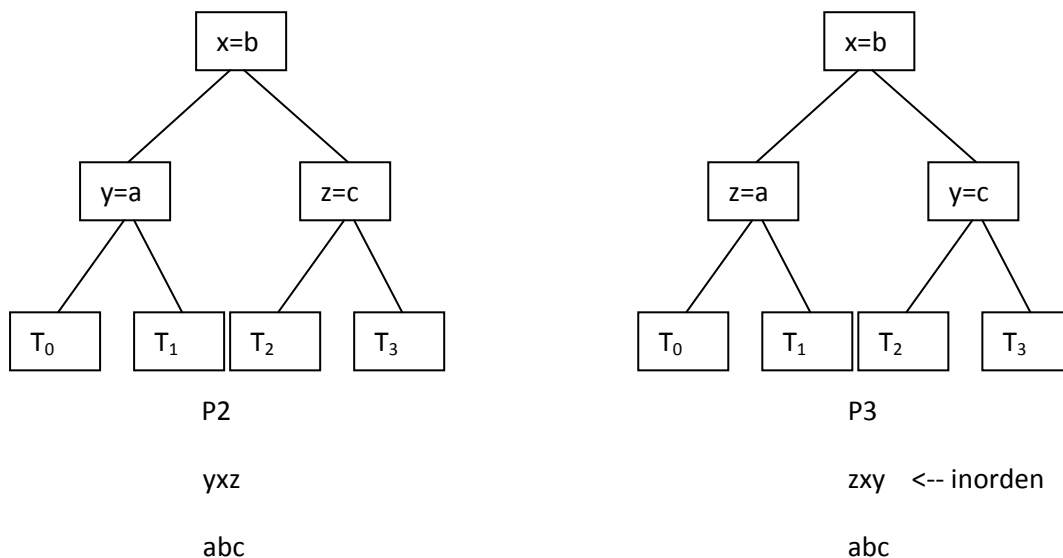
abc

abc

#+END_EXAMPLE

2. Dobles ('double rotation'). Cuando $b=x$. (P2 y P3).
 - 2.1. Se rota 'x' sobre 'y'. Un hijo de 'x' pasa a ser hijo de 'y' (T_1 en P2 y T_2 en P3) poniéndose 'y' en lugar de dicho hijo de 'x'.
 - 2.2. Se rota 'x' sobre 'z'. Un hijo de 'x' (T_2 en P2 y T_1 en P3) pasa a ser hijo de 'z' poniéndose 'z' en lugar de dicho hijo de 'x'.

#+BEGIN_EXAMPLE



#+END_EXAMPLE

ERRORES: [Transparencia 8]: En la segunda doble rotación los subárboles deberían estar numerados de izquierda a derecha igual que en la primera doble rotación: T_0, T_1, T_2, T_3 , al igual que en [Libro, p448].

En términos de implementación, las "rotaciones" involucran simplemente el intercambio de referencias (valores de atributos) de los nodos. Por ejemplo, la rotación doble de P2 involucra las siguientes asignaciones (en pseudocódigo):

```
#+BEGIN_EXAMPLE
hijo_derecho(y) = hijo_izquierdo(x)
hijo_izquierdo(x) = y
hijo_izquierdo(z) = hijo_derecho(x)
hijo_derecho(x) = z
#+END_EXAMPLE
```

Ejercicio: Escribir el pseudocódigo del resto de rotaciones.

Las rotaciones producen árboles equilibrados pues 'x' e 'y' son los hijos de mayor altura. **En la inserción, el desequilibrado es un efecto local** [Libro, p447].

La complejidad de la inserción es $O(\log n)$:

- La búsqueda de la entrada en el árbol es $O(\log n)$.
- Los nodos de un árbol AVL llevan cuenta de su altura. Al insertar un nodo se comprueba el equilibrio de los nodos en el camino del nodo insertado a la raíz (sólo la rama que ha "crecido") hasta llegar o bien a la raíz (sin producirse desequilibrado) o bien a 'z' (donde se ha producido un desequilibrado). En el caso peor 'z' es la raíz y el recorrido hasta 'z' es $O(\log n)$.
- Se realizan las rotaciones que son meros intercambios de valores con la [Clase LinkedBinaryTree<E>](#). El coste es $O(1)$.
- Total: $O(2 \cdot (\log n) + 1)$, es decir, $O(\log n)$.

Ejercicio: ¿Cómo afectaría a las rotaciones que se permitieran entradas con claves repetidas (árboles AVL para diccionarios con dominio ordenado)?

Borrado

Si la clave está en el árbol se borra la entrada como en un árbol binario de búsqueda [10.1.2 Update Operations](#).

Puede producirse un desequilibrado después de invocar 'removeExternal'.

- Sea 'z' el primer nodo desequilibrado encontrado en el camino desde el nodo borrado hasta la raíz.
- Sea 'y' el hijo de 'z' de mayor altura.
- Sea 'x' el hijo de 'y' de mayor altura. Si los hijos de 'y' tienen la misma altura se elige el hijo en el mismo "lado" que 'y' para evitar la doble rotación (combinaciones P1 y P4).
- Se aplica 'restructure(x)' como en la inserción.
- Ahora el equilibrio sí puede desequilibrar nodos ancestros. Hay que seguir equilibrando en el camino hasta llegar, en el caso peor, a la raíz. El coste es $O(\log n)$.

Ejercicio: ¿Podríamos haber usado en la inserción el mecanismo de elección de 'x' usada en el borrado?

Complejidad para árboles AVL

Complejidad de los métodos:

get	$O(\log n)$	Búsqueda binaria en árbol equilibrado
put	$O(\log n)$	Búsqueda binaria + equilibrado
remove	$O(\log n)$	Búsqueda binaria + equilibrado(s)
entrySet	$O(n)$	Recorrido en inorden

ERRORES [Transparencia 11]: En el caso de 'put' no hay un "restructuring up the tree".

10.2.2 Java Implementation

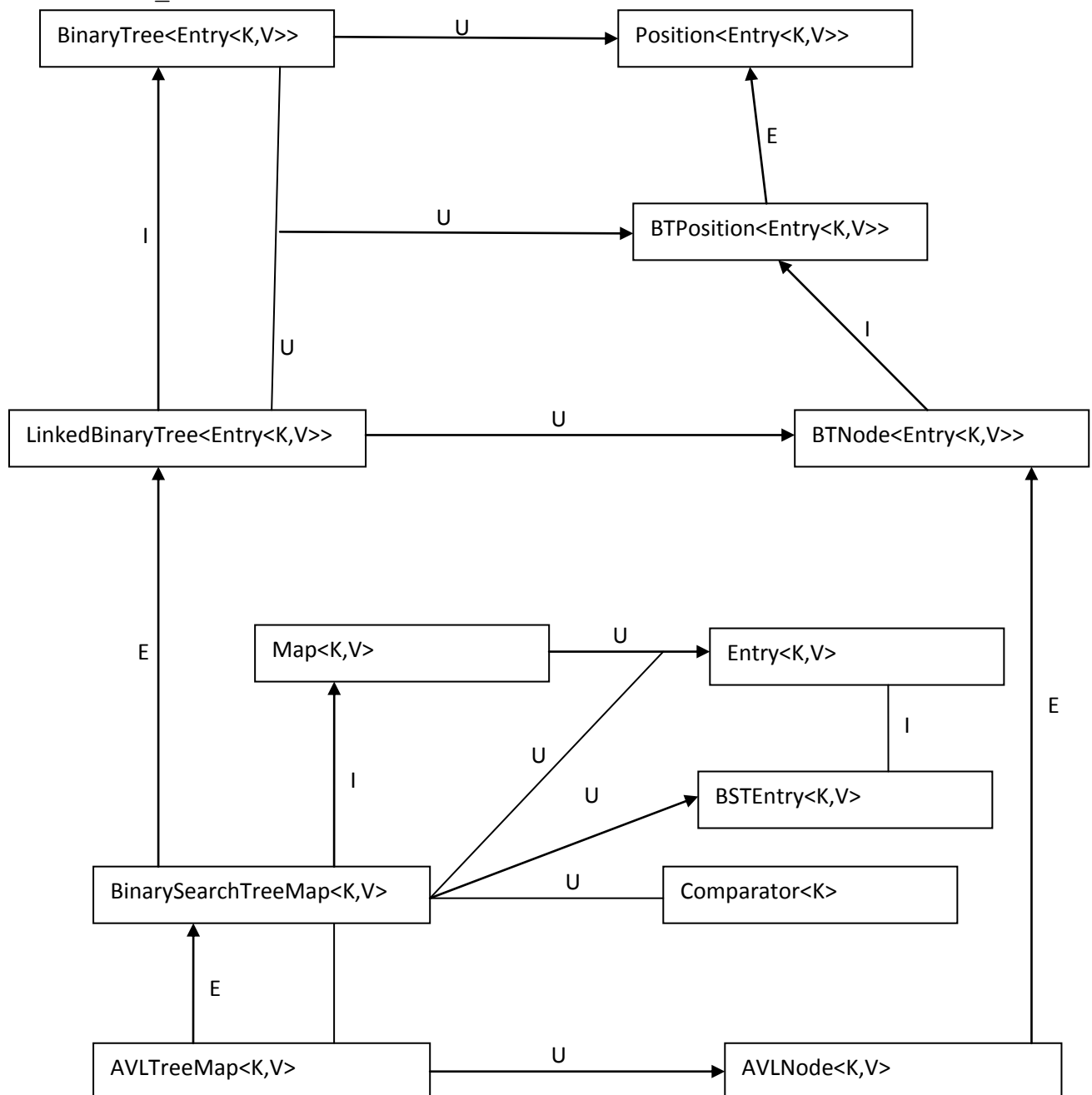
Código: AVLTreeMap.java.

La clase 'AVLTreeMap<K,V>' extiende la clase 'BinarySearchTreeMap<K,V>' (árboles binarios de búsqueda [10.1.3 Java Implementation](#)) e implementa el interfaz 'Map<K,V>' (funciones finitas [9.1.1 The Map ADT](#)).

Como en [10.1 Binary Search Trees](#), se asume dominio ordenado y la [Representación mediante árboles binarios](#).

Diagrama de clases involucradas

#+BEGIN_EXAMPLE



#+END_EXAMPLE

U: Usa. I: Implementa. E: Extiende.

La clase anidada 'AVLNode<K,V>', que implementa los nodos de árboles AVL, extiende la [Clase BTreeNode<E>](#), que implementa los nodos de árboles binarios:

- Añade el atributo 'height', altura del nodo.
- El constructor por defecto crea un nodo vacío (todos los atributos a null) y el segundo constructor invoca el constructor de 'BTreeNode' (vía 'super') y calcula la altura del nodo. Este constructor toma como argumento nodos que implementan el [Interfaz BTPosition<E>](#), de ahí los downcastings a 'AVLNode' en el código que calcula la altura. Las alturas de los ancestros del nodo insertado serán actualizadas por los métodos de inserción y borrado.
- Los métodos 'getHeight' y 'setHeight' son respectivamente un "getter" y un "setter" para el atributo 'height'.

ERRORES: Los downcastings a 'AVLNode' en el código de la clase podrían lanzar una excepción no recogida si el nodo, del que sólo sabemos que implementa 'BTPosition', no es un objeto de clase 'AVLNode'. Se debería definir e invocar un método 'checkNode' que comprobase la validez del nodo, como se hace en otras clases que hemos visto. El segundo constructor, por ejemplo, debería invocar 'checkNode' antes de 'super'.

Se sobrescribe el método 'createNode' de la [Clase LinkedBinaryTree<E>](#) para que los nodos construidos por los métodos heredados de dicha clase (que implementa árboles binarios) sean de clase 'AVLNode' y no de clase 'BTreeNode'. Véase [Críticas a la implementación AVLTreeMap](#).

Los métodos 'height' y 'setHeight' de 'AVLTreeMap' (el árbol) llaman a los métodos 'getHeight' y 'setHeight' de 'AVLNode' (el nodo). Los primeros toman un objeto que implementa 'Position<Entry<K,V>>'. Se definen estos métodos para abreviar los pasos necesarios de downcasting y cálculo de la altura.

Ejercicio: El método 'setHeight' de 'AVLNode' es un simple "setter". ¿Debería haberse definido el 'setHeight' de 'AVLNode' como el 'setHeight' de 'AVLTreeMap' y no tener que definir este último en la clase 'AVLTreeMap'?

El método 'isBalanced' indica si un nodo está equilibrado usando el factor de equilibrio [Representación mediante árboles binarios de búsqueda](#).

El método 'tallerChild' indica cuál de los dos hijos de un nodo tiene mayor altura. Es usado por 'rebalance' para elegir los nodos involucrados en las rotaciones (Ver [10.2.1 Update Operations](#)).

El método 'rebalance' recalcula alturas en el camino a la raíz e invoca 'restructure' cuando encuentra un nodo desequilibrado. Recuerdese que el método 'restructure' está (erróneamente) en la clase 'BinarySearchTreeMap'. Véase [10.1.3 Java Implementation](#).

Se sobrescriben (refinan) los métodos 'put' y 'remove' de la clase 'BinarySearchTreeMap' que ahora deben realizar equilibrados. Primero se invocan los métodos 'put'/'remove' de

'BinarySearchTreeMap' (vía 'super') y luego se invoca 'rebalance' sobre el [nodo 'actionPos'](#), atributo protegido heredado de la clase padre 'BinarySearchTreeMap'.

Críticas a la implementación AVLTreeMap.

Las clases que implementan los nodos de los árboles binarios ('BTNode') y de los árboles AVL ('AVLNode') implementan ambas el interfaz 'Position'. La clase 'AVLTreeMap' extiende 'BinarySearchTreeMap' que a su vez extiende la clase padre 'LinkedBinaryTree'. En Java los constructores no pueden sobrescribirse. La clase 'LinkedBinaryTree' tiene que construir nodos mediante un método ordinario sobrescribible ('createNode') para que los métodos heredados de ésta por 'AVLTreeMap' ('addRoot', 'insertLeft' e 'insertRight') construyan objetos 'AVLNode' y no 'BTNode'.

El [nodo 'actionPos'](#) es un atributo de la clase padre 'BinarySearchTreeMap'.

El método 'restructure' debe estar en 'AVLTreeMap' y no en 'BinarySearchTreeMap'.

Explorar como posible solución el uso de algún patrón de diseño (Model-View-Controller, Observer) para rediseñar árboles binarios, de búsqueda y AVL.

Viernes 20-05-2011

10.4 (2,4) Trees

Material

Libro: Sección 10.4.

Transparencias: 24trees.pdf.

Código: no hay código para este tema.

Repasar [10.1 Binary Search Trees](#).

Motivación de árboles multicamino de búsqueda y árboles (2,4)

'Multi-way' = multi-camino = posibilidad de tener más de dos hijos. Los árboles multicamino de búsqueda se utilizan para implementar funciones finitas o diccionarios, ambos con dominio ordenado. El libro se centra únicamente en funciones finitas. Este guión también.

Los árboles multicamino de búsqueda son una generalización de los árboles binarios de búsqueda. Pueden entenderse como "árboles generales de búsqueda" cuyos nodos internos tienen ≥ 2 hijos y almacenan colecciones de entradas ordenadas por clave.

El objetivo es reducir la altura del árbol ampliando su anchura, sin renunciar a la complejidad logarítmica de las operaciones.

Los árboles (2,4) son un caso particular de árbol multicamino de búsqueda en el que los nodos internos tienen entre 2 y 4 hijos y el árbol está "equilibrado" en un sentido más estricto que en los árboles binarios (Ver [Binary Trees: Terminología y Definiciones](#)).

10.4.1 Multi-Way Search Trees

Definición de d-nodo y d-árbol

En [General Trees: Terminología y Definiciones](#) se define el **grado de un nodo** como el número de hijos del nodo.

Llamamos **d-nodo** a un nodo de grado 'd' (0-nodo, 1-nodo, 2-nodo, etc).

Llamamos **d-árbol** a un árbol general no vacío cuyos nodos internos tienen mínimo 2 y máximo d hijos. El mínimo posible son los 2-árboles, que son los árboles binarios propios (Ver [Binary Trees: Terminología y Definiciones](#)).

Representación e implementación de árboles multicamino de búsqueda

Árboles vacíos y nodos externos

Al igual que ocurre con la [Representación mediante árboles binarios](#) de los árboles binarios de búsqueda, **los árboles multicamino de búsqueda vacíos y los nodos externos son representados por nodos con elemento a null**. Las razones son las mismas que las expuestas para los árboles binarios de búsqueda.

En las figuras de árboles multicamino de búsqueda del libro y las transparencias los nodos externos se dibujan como cuadrados vacíos.

Nodos internos y búsqueda

[Transparencias 2-4]

Todo d-nodo interno 'v' almacena una función finita con dominio ordenado que denotamos 'M(v)' ("map" de 'v') y que guarda d-1 entradas. (Esto explica por qué $d \geq 2$: para no tener nodos internos sin entradas.) Alternativamente, decimos que si hay 'e' entradas en la función finita entonces el nodo tiene 'e+1' hijos.

Si 'd' es siempre pequeño la función finita con dominio ordenado puede estar implementada mediante un vector ('array') o mediante una lista 'ArrayIndexList'. Se prefiere búsqueda binaria $O(\log d)$ pero cuando 'd' es suficientemente pequeño puede hacerse búsqueda lineal $O(d)$, tener el vector o lista desordenado, o usarse una 'NodePositionList'. "Suficientemente pequeño" significa para valores de 'd' tal que la diferencia entre 'log d' y 'd' no sea significativa.

Si 'd' es un valor fijo constante entonces no depende del número total de entradas en el árbol y realmente las complejidad de las operaciones de la función finita del nodo serán $O(1)$.

Se puede crear una implementación de funciones finitas específica para árboles multicamino de búsqueda que use un vector, realice búsqueda binaria, etc.

Los árboles multicamino de búsqueda no se implementan usando árboles generales [7.1 General Trees](#). Se guardan los hijos (concretamente una referencia a sus nodos raíces) como parte de la información asociada a las claves. Así, en un d-nodo 'v' del árbol, la función finita 'M(v)' almacena las siguientes entradas:

$$(k_1, (x_1, v_1)) \dots (k_{(d-1)}, (x_{(d-1)}, v_{(d-1)})) (sup, (null, v_d))$$

Cada entrada tiene una clave k_i y un valor que es a su vez una entrada con la información x_i y con el subárbol (referencia al nodo raíz) v_i donde están las claves **menores** que k_i .

La entrada 'sup' no es una entrada legal, no guarda información. Se usa para facilitar la búsqueda. Solo hay $d-1$ entradas legales en el d -nodo. El valor 'sup' es el supremo de las claves: para toda clave k se cumple $k < \text{sup}$. El tipo de las claves debe tener un supremo o un valor de bandera que lo simule.

Para toda entrada $(k_i, (x_i, v_i))$, las claves de v_i son menores que k_i (esto también se cumple para la entrada 'sup') y para $1 < i < d$ las claves en v_i están estrictamente entre $k_{(i-1)}$ y k_i . Las claves en v_d son mayores que $k_{(d-1)}$, están entre $k_{(d-1)}$ y 'sup'.

Búsqueda: se empieza buscando una clave k en el nodo raíz:

- Si el nodo es externo (elemento a null, no hay función finita) la clave no está en el árbol.
- Si el nodo es interno, se busca la clave en la función finita con el método 'ceilingEntry' (Ver [9.3 Ordered Maps](#) para obtener la entrada $(k_i, (x_i, v_i))$ tal que k_i es la menor clave que cumple $k \leq k_i$. Si $k = k_i$ entonces se devuelve la información x_i . Si $k < k_i$ entonces se busca recursivamente en v_i . (Al llegar a la entrada 'sup' se cumple $k < \text{sup}$ y se busca en $v_{(d-1)}$.)

Complejidad de la búsqueda:

- La complejidad de la búsqueda en el árbol depende de la complejidad de la búsqueda en los nodos, que en caso peor hemos visto que es $O(d)$. Hay que multiplicar ese factor por la altura del árbol $O(d \cdot h)$, ya que en el caso peor la clave no está en el árbol y se busca desde la raíz hasta un nodo externo.
- Como d es constante, tenemos $O(h)$ como en árboles binarios de búsqueda (Ver [Complejidad para árboles binarios de búsqueda](#)).
- Si el árbol está equilibrado entonces tenemos $O(\log n)$ como en árboles AVL (Ver [Complejidad para árboles AVL](#)).

ERRORES: [Libro, 'Definition of a Multi-way Search Tree', p465] La definición dice " $k_1 \leq \dots \leq k_{(d-1)}$ " permitiendo claves repetidas. Esto sólo es válido para diccionarios. En [Libro, 'Searching in a Multi-way Tree', p467] se dice correctamente " $k_{(i-1)} < k < k_i$ ". También la [Transparencia 2] es correcta si 'between' se entiende en sentido estricto.

[Libro, 'Proposition 10.7', p467]: Un árbol multicamino de búsqueda con ' n ' entradas tiene ' $n+1$ ' nodos externos.

Ejercicio: Demostrar la proposición anterior basándose en la definición de árbol multicamino de búsqueda.

Recorrido en inorden de árboles multicamino de búsqueda
Se generaliza el recorrido en inorden de árboles binarios:

```
#+BEGIN_EXAMPLE

  Inorden(v) :

  Para i de 1 hasta d-1:
    InOrden(v_i)
    Se hace algo con k_i y x_i de v.

  InOrden(v_d)

#+END_EXAMPLE
```

Definición de árboles (2,4)

[Transparencias 5-6]

Los árboles (2,4) son árboles multicamino de búsqueda que cumplen:

- Los nodos internos tienen de 2 a 4 hijos. (Almacenan de 1 a 3 entradas, 'd' es constante y pequeño). El [Libro, p469] llama a esta propiedad 'size property'.
- Los nodos externos tienen la misma profundidad. (El árbol está totalmente equilibrado: los nodos de un nivel tienen la misma altura). El [Libro, p469] llama a esta propiedad 'depth property'.

[Transparencia 5]: La altura de un árbol (2,4) es $O(\log n)$.

10.4.2 Update Operations for (2,4) Trees

Inserción

[Transparencias 7-9]

Se busca la clave como se ha descrito en [Nodos internos y búsqueda](#). Si está en algún nodo se actualiza la información. Si no está, la búsqueda termina en un nodo externo con padre 'v'. Se mete la nueva entrada en 'v' con la clave, la información y un nodo externo.

Posibilidad de **desbordamiento** ('overflow') si 'v' es un 4-nodo (3 entradas y 4 hijos): al insertar sería un 5-nodo (4 entradas y 5 hijos) con lo que se incumple la 'size property'.

Resolución de desbordamiento: **desdoblar** ('split') el 5-nodo en un 3-nodo y un 2-nodo:

Tras insertar la nueva clave en M(v) tenemos 4 entradas (legales) y 5 hijos:

$$(k_1, (x_1, v_1)) (k_2, (x_2, v_2)) (k_3, (x_3, v_3)) (k_4, (x_4, v_4)) (\text{sup}, (\text{null}, v_5))$$

1. Las dos primeras entradas a un 3-nodo (2 entradas y 3 hijos):

$(k_1, (x_1, v_1)) (k_2, (x_2, v_2)) (\text{sup}, (\text{null}, v_3))$

Se reemplaza este nodo por 'v' en el padre de 'v'.

1. Se sube al padre la entrada $(k_3, (x_3, w))$ donde 'w' es el 2-nodo creado en el siguiente punto.
2. La cuarta entrada (y la ficticia) a un 2-nodo (1 entrada y 2 hijos):

$(k_4, (x_4, v_4)) (\text{sup}, (\text{null}, v_5))$

El segundo punto puede producir desbordamiento en el padre, con lo que habría que desdoblar el padre. Podría haber desbordamientos encadenados en el camino hasta la raíz. Hay que crear un nodo raíz nuevo si se produce desbordamiento en la raíz, ya que el nodo raíz se desdobla en dos nodos nuevos y hay que subir una entrada a un padre.

La complejidad en caso peor de la inserción es $O(h)$ con $h = \log n$, pues el árbol cumple la 'depth property'. Desdoblar hasta la raíz es $O(h)$ pues se desdoblan sólo nodos en el camino hasta la raíz:

Buscar $O(h)$ + Insertar en nodo $O(1)$ + Desdoblar hasta raíz $O(h)$ = $O(h)$

Borrado

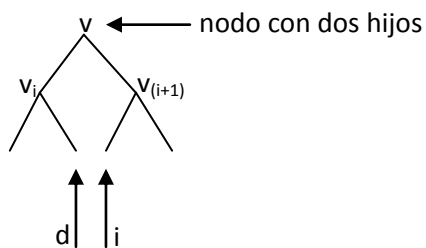
[Transparencias 10-13]

Se busca la clave como se ha descrito en [Nodos internos y búsqueda](#). Si no está (se llega a un nodo externo) no hay nada que borrar. Si está, la búsqueda termina en un nodo interno 'v'. Como en los árboles binarios de búsqueda, hay que mover una entrada de un nodo frontera. El número de entradas en 'v' queda constante. Si ya estamos en un nodo frontera, se elimina la entrada sin más.

Sea k_i la clave a borrar en nodo no frontera 'v':

- El nodo frontera a elegir será o bien el nodo más a la derecha 'der' en el subárbol ' v_i ' que contiene las claves menores que ' k_i ' o bien el nodo más a la izquierda 'izq' en el subárbol ' v_{i+1} ' que contiene las claves mayores que ' k_i '.

#+BEGIN_EXAMPLE



#+END_EXAMPLE

Como en los árboles binarios de búsqueda [10.1.2 Update Operations](#) se puede tener borrado asimétrico (se elige siempre el mismo nodo frontera) o asimétrico (se alterna entre 'der' e 'izq' según algún criterio fijo o aleatorio).

- La clave a elegir será la mayor (ojo, 'sup' no es una clave) de 'der' (que es la anterior a 'k_i' en un recorrido en inorden del árbol) o la menor de 'izq' (que es la siguiente a 'k_i' en un recorrido en inorden del árbol).
- Se mueve la entrada con la clave elegida desde el frontera a 'v'.

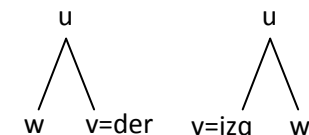
Al quitar una entrada en el nodo frontera, si dicho nodo es un 2-nodo se puede producir un **déficit** ('underflow'). Sabemos por la 'depth property' que el nodo frontera tiene que tener hermanos. Nos fijamos en el hermano inmediato [Libro, "immediate sibling", p474].

Hay dos posibilidades de elección de hermano inmediato según el método de elección del nodo frontera:

1. Si se eligió 'der' entonces el hermano inmediato es el primer nodo a la izquierda de 'der'.
2. Si se eligió 'izq' entonces el hermano inmediato es el primer nodo a la derecha de 'izq'.

Sea 'v' el nodo frontera, 'w' el hermano inmediato elegido, 'u' el padre de ambos:

#+BEGIN_EXAMPLE



#+END_EXAMPLE

Hay dos formas de solucionar el déficit:

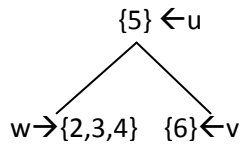
- **Fusión** (cuando 'w' es un 2-nodo): fusionar 'w' con 'v' (ahora hay un único nodo) moviendo además una clave de 'u' al nuevo nodo fusionado. ¿Qué clave? La mayor o la menor de 'u' dependiendo de si el nodo frontera es 'der' o 'izq' respectivamente. Se puede producir déficit en 'u' con lo que habría que reaplicar una de las soluciones (fusión o transferencia), pudiéndose producir déficit hasta llegar a la raíz.
- **Transferencia** (cuando 'w' es 3-nodo o 4-nodo):
 1. Se mueve un nodo externo de 'w' (hermano) a 'v' (frontera)
 2. Se mueve una clave de 'u' (padre) a 'v' (frontera).
 3. Se mueve una clave de 'w' (hermano) a 'u' (padre).

¿Qué clave? La mayor o la menor de 'u' dependiendo de si el nodo frontera es 'der' o 'izq' respectivamente.

Después de una transferencia no se produce otro déficit. El número de entradas en 'u' permanece constante.

ERRORES: [Libro, p474] habla de "immediate sibling" sin precisar cuál elegir. También habla de utilizar fusión cuando el frontera tiene un único hermano (que podría ser 3- o 4-nodo) lo cual no es correcto. Contraejemplo:

#+BEGIN_EXAMPLE



#+END_EXAMPLE

Aplicando fusión habría desbordamiento en 'w' y déficit en 'u'. La complejidad en caso peor de la fusión y/o transferencia es $O(h)$.

Complejidad en árboles (2,4)

La complejidad de la búsqueda, inserción y borrado es $O(h)$. Por la 'depth property' se tiene $h = \log n$, con lo que la complejidad es $O(\log n)$.

Viernes 27-05-2011

14.2 External Memory and Caching

Material

Libro: Sección 14.2.

Transparencias: ext_memory_caching.pdf.

Código: no hay código para este tema.

Algunas notas a las transparencias

Como vimos en [4.2 Analysis of Algorithms](#), es correcto ignorar el coste de acceso a memoria en el diseño de algoritmos y TADs. Éstos se pueden optimizar para arquitecturas específicas.

Proximidad temporal ('temporal locality'): las direcciones de memoria accedidas recientemente en el tiempo tienen alta probabilidad de ser accedidas de nuevo en el futuro cercano. La proximidad temporal está vinculada al comportamiento dinámico del código.

Proximidad espacial ('spatial locality'): las direcciones de memoria cercanas en el espacio a las accedidas recientemente en el tiempo tienen alta probabilidad de ser accedidas en el futuro cercano. La proximidad espacial está vinculada a la estructura estática (texto) del código.

Los bucles 'while' y 'for' con pocas líneas son ejemplos de código con proximidad espacial y temporal.

Líneas de cache: bloques de memoria interna a memoria cache.

Páginas: bloques de memoria externa a memoria interna.

FIFO (cola) vs LRU (cola con prioridad): la inserción en una cola con prioridad tiene mayor coste que en una cola FIFO (Ver [8 Priority Queues](#)). Estudios empíricos sugieren que LRU es superior a FIFO.

14.3 External Searching and B-Trees

Material

Libro: Sección 14.3.

Transparencias: no usaremos transparencias para este tema.

Código: no hay código para este tema.

Repasar: [10.4 \(2,4\) Trees](#).

ERRORES: En el texto del libro se mencionan los diccionarios cuando realmente la discusión sólo se centra en funciones finitas.

Motivación de los árboles (a,b) y B

Los TADs y algoritmos para funciones finitas con dominio ordenado no toman en cuenta que los datos (casillas de un vector, nodos de listas enlazadas, nodos de árboles) no pueden estar todos simultáneamente en memoria principal. La gran mayoría estará en memoria externa y se desea minimizar el coste de acceso.

Los árboles (a,b) generalizan los (2,4) en el número mínimo 'a' y máximo 'b' de hijos. Las operaciones de búsqueda, inserción y borrado son una generalización directa, variando sólo el grado de los nodos.

Los árboles B son un caso general de árboles (a,b) en el que 'a' y 'b' se eligen de forma que los nodos internos ocupen eficientemente (en términos de espacio) **bloques** de memoria externa (de ahí la "B" de su nombre).

14.3.1 (a,b) Trees

[Libro, p679]:

- 'Size property': exceptuando la raíz, los nodos internos tienen entre 'a' y 'b' hijos (entre 'a-1' y 'b-1' entradas) con $2 \leq a \leq (b+1)/2$. Esta última propiedad se necesita para realizar desdoblamientos ('split') [10.4.2 Update Operations for \(2,4\) Trees](#) de forma que se mueva al padre del nodo desbordado la clave del medio de éste último.
- 'Depth property': los nodos externos tienen la misma profundidad.

[Libro, p679]: La altura de un árbol (a,b) con n entradas tiene un valor asintótico entre $\log_b n$ y $\log_a n$.

14.3.1 B-Trees

Los árboles B y sus variantes (B^* , B^+ , etc) son la mejor representación conocida para almacenar árboles en memoria externa.

Uno o más nodos internos se almacenan en bloques de memoria externa: b es proporcional al tamaño B del bloque.

ERRORES: En [Libro, Sección 14.3.2] B es el tamaño del bloque mientras que en '[Libro, 'Some Inefficient External-Memory Dictionaries', p678] B es el número de nodos de una lista (la que implementa la función finita del nodo interno) que caben en un bloque de memoria externa.

Se elige $a = b/2$ (división entera) para tener nodos al menos medio llenos.

[Libro, 'Proposition 14.2', p681] La búsqueda, inserción y el borrado requieren $O(\log_B n)$ transferencias de bloques y usan $O(n/B)$ bloques.